UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

20|21

Trabajo fin de grado

Desarrollo de interfaces para White boxes

Alejandro Sánchez Sanz

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

Campus Internacional excelencia UAM CSIC+

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

Desarrollo de interfaces para White boxes

Autor: Alejandro Sánchez Sanz
Tutor: Víctor López Álvarez

junio 2021

Alejandro Sánchez Sanz
Desarrollo de interfaces para White boxes

Alejandro Sánchez Sanz
C\ Francisco Tomás y Valiente Nº 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A mis padres, Alejandro y Paloma. Por su inmenso apoyo y cariño, que me han llevado a ser como soy.

Some people want it to happen, some wish it would happen, others make it happen.
(Algunas personas quieren que algo ocurra, otras sueñan con qué pasará, otras hacen que suceda)

Michael Jordan

Agradecimientos

En primer lugar, me gustaría agradecer a mi tutor, Víctor López, toda la guía y ayuda recibida durante este año en el que hemos trabajado juntos. Eso sí, todavía nos queda pendiente esa partida de ajedrez.

A continuación, querría agradecer a las personas que me han orientado y ayudado a entender los proyectos *OpenSource* tratados durante este TFG. Ellos son Wataru Ishida (*NTT Electronics*), para el proyecto de *Goldstone*, y especialmente Šimon Mišenčík (*FRINX*) para la herramienta FRINX UniConfig.

También me gustaría agradecer a mi familia, en especial a mis padres y hermanos, toda la confianza depositada en mí y todo el apoyo recibido durante toda mi etapa universitaria. Han sido muchas horas sentado estudiando y programando enfrente del ordenador. Muchas noches trabajando hasta entrada la madrugada, donde aparecía alguno de ellos para decirme que no me acostase demasiado tarde, a pesar de que sabía que sentía que debía continuar trabajando. Todo ese esfuerzo, ha merecido la pena.

A mi novia, Paula, que siempre ha creído en mí durante todos estos meses, incluso cuando yo mismo tenía dudas. Su cariño no tiene precio, y los consejos que me ha dado tras haber escrito ella ya un TFG y un TFM, han sido muy valiosos.

Por último, no puedo olvidar agradecer a todos mis compañeros de clase. Durante estos años hemos podido aprender, reír, sufrir, disfrutar y superar obstáculos juntos. Estoy muy agradecido por haber compartido estos años con todos ellos y, como tantas veces me han escuchado gritar:

¡Viva la Doble!

Resumen

En la comunidad de las redes de comunicaciones actual, existe cada vez mayor interés por las soluciones desagregadas de los elementos de red, donde un elemento cada vez más importante son las *White-Boxes*. Estas necesitan tener interfaces estándar para poder integrarse de forma sencilla en las redes del operador. Este TFG realizará extensiones a los protocolos y validación de los mismos para facilitar el despliegue de estas soluciones de red.

En concreto, se probarán primero dos NOS (*Network Operating System*): Goldstone y SONiC. Tras escoger el segundo, se implementará una *cli-unit* (unidad de traducción de CLI) como contribución al proyecto *OpenSource* de FRINX UniConfig, para facilitar las soluciones basadas en *White-Boxes* con ese NOS. Para poder probar lo implementado, se construirá un entorno de red virtual basado en contenerización (con Docker). Este entorno lo formarán dos *switches* (conmutadores de red) con el sistema operativo SONiC, que se comunicarán entre ellos. Serán accesibles por SSH, por lo que será posible conectarse con la herramienta FRINX UniConfig para configurarlos a través del empleo de la unidad de traducción de CLI implementada.

Palabras clave

Redes definidas por Software, Cajas Blancas, Virtualización, Docker, SONiC, FRINX UniConfig, Proyecto de código abierto

Abstract

In the actual network community, it exists a higher interest for solutions based on disaggregated network elements, where an element that is increasing its importance, are the White-Boxes. They need to have standard interfaces, in order to be integrated in an easy way in operator networks. This Bachelor Thesis will make extensions and validation of the protocols to ease the deploy of these network solutions.

Specifically, two NOS (Network Operating System) will be tested at the beginning: Goldstone and SONiC. After choosing the second one, a *cli-unit* will be implemented as a contribution to the Open-Source project FRINX UniConfig, in order to ease the solutions based on White-Boxes with that NOS. To be able to test the implemented software, a virtual network environment based on containerization (with Docker) will be built. This environment will be composed of two switches running SONiC, that will communicate with each other. They will be accessible via SSH, hence it will be possible to connect to them with the tool FRINX UniConfig, to configure them through the usage of the implemented CLI translation unit.

Keywords

Software-Defined Networks, White-Boxes, Virtualization, Docker, SONiC, FRINX UniConfig, Open-Source project

ÍNDICE

Listas

Lista de figuras

1 Introducción

1.1. Motivación

En el mundo tecnológico existen multitud de dispositivos distintos, creados por distintas compañías. Originalmente, lo más extendido era que cada una generase un hardware que funcionase solo con el software del mismo fabricante, lo cual condiciona bastante a los consumidores. Es por eso que, en los últimos años, cada vez más fabricantes apuestan por las soluciones desagregadas, que permitan a los usuarios combinar hardware y software de un modo más flexible.

Posiblemente nos vengan a la mente los ordenadores, pues hoy en día podemos instalar distintos sistemas operativos (software) en un ordenador (hardware) según nos convenga, sin necesidad de que el fabricante sea el mismo. Pero cada vez hay más interés sobre este mismo tema en la comunidad de las redes de comunicaciones, tratando de proporcionar vías para la desagregación de hardware y software. Esto permitiría crear soluciones más adecuadas a cada situación, pero también aumentaría la complejidad del sistema. Aquí entran en juego las soluciones basadas en redes definidas por software (*software-defined networks (SDN)*), que ofrecen una visión estándar de los elementos de red, de modo que los operadores de red pueden tener un diseño extremo a extremo (*end-to-end*) más claro.

También juegan un papel importante las *White-Boxes* (cajas blancas). Estas son diseños de hardware que permiten a los usuarios instalar cualquier software para operar la red. Por tanto, se emplean en las soluciones híbridas ya mencionadas donde el software y hardware pertenecen a fabricantes distintos. Sin embargo, esto hace que sea necesario tener interfaces estándar para poder integrarlos de forma sencilla en las redes del operador. También, que existan protocolos para configurar y modelos para describir dichos dispositivos de un modo común, como busca *OpenConfig*.

Hoy en día, los proyectos *Open Source* (código abierto) tienen una gran relevancia en la comunidad tecnológica. Por medio de todos los desarrolladores que contribuyen, estos proyectos crecen y mejoran de calidad a gran velocidad. Además, son una fuente pública de conocimiento y que permite a las compañías su uso e integración en sus dispositivos. Por estas y otras razones, cada vez más gente apuesta por estos proyectos. Muchos de los elementos mencionados en los párrafos anteriores son proyectos de código abierto, por lo que están en constante investigación y crecimiento gracias a la

comunidad y a sus necesidades.

1.2. Objetivos

Los objetivos del trabajo son los siguientes:

- Describir las soluciones basadas en tecnologías abiertas.
- Conocer herramientas para facilitar el desarrollo de interfaces abiertas.
- Desarrollar el entorno de pruebas para validar el funcionamiento.
- Demostrar el funcionamiento de la interfaz desarrollada.

1.3. Metodología

Este trabajo ha comenzado por una labor de investigación y documentación de las redes definidas por software, las *White-Boxes* y distintos protocolos de configuración de dispositivos de red. Esta parte ha llevado gran parte del tiempo, ya que estos son conceptos relativamente recientes, sobre los que no existe tanta literatura todavía.

Una vez adquirido el conocimiento necesario, se comenzaron a probar dos sistemas operativos para tratar de desarrollar alguna interfaz sobre ellos. Se diseñaron y construyeron entornos virtualizados para el proceso. Una vez se vio que uno tenía una comunidad más grande y existían proyectos de código abierto donde poder contribuir de un modo más claro, se decidió continuar con él.

Finalmente, una vez implementada la contribución, para demostrar su funcionamiento se preparó una ampliación del entorno virtualizado basado en contenerización y se realizó una *demo*.

1.4. Estructura del documento

Este proyecto está estructurado de la siguiente manera:

- **Capítulo 2: Estado del arte.** En este capítulo se explicarán muchos elementos y conceptos sobre los que tratará el resto del trabajo. Se hablará de los dispositivos de red y algunos protocolos de configuración, especialmente *OpenConfig*, al ser el común que tratan de soportar todos los fabricantes que quieren proporcionar soluciones basadas en *White-Boxes*. Se explicará más en detalle qué son estas últimas, indicando sus ventajas. Después se hablará de dos proyectos *OpenSource*: SONiC, un sistema operativo de redes (*Network Operating System (NOS)*), y FRINX UniConfig, una herramienta que trata de gestionar el estado configuracional y recuperar el estado operacional de dispositivos de red físicos y virtuales.
- **Capítulo 3: Diseño.** En este capítulo comenzaremos hablando del concepto de contenerización y de dos ejemplos (Docker y Kubernetes), pues serán base para los diseños de los que hablaremos después. A continuación, explicaremos los diseños de entornos virtualizados basados en Goldstone y SONiC, explicando más en detalle la ampliación final del segundo.

- **Capítulo 4: Desarrollo.** En este capítulo explicaremos el material construido o implementado para crear y configurar los entornos diseñados en el capítulo anterior. Primero, hablaremos sobre la configuración de Goldstone y los pasos que tuvimos que realizar para lanzar su entorno virtual en local. Después, se hablará del desarrollo de la unidad de traducción (*translation-unit*) implementada para SONiC, que servirá como contribución al proyecto *OpenSource* de FRINX. Por último, explicaremos la ampliación del entorno virtual basado en SONiC que se ha realizado con el fin de probar la unidad de traducción implementada para FRINX, y demostrar que funciona correctamente.
- **Capítulo 5: Pruebas y resultados.** En este capítulo comenzaremos ilustrando las pruebas realizadas tras el desarrollo del proyecto, que tratan de demostrar el correcto funcionamiento del mismo. Finalmente, se mostrará el resultado de la contribución *OpenSource* realizada.
- **Capítulo 6: Conclusiones y trabajo futuro.** En este capítulo final se revisarán los objetivos definidos en la introducción para justificar que se han cumplido. Se explicarán también las conclusiones obtenidas durante el desarrollo del trabajo, así como los conocimientos y heramientas que han ayudado a la elaboración del mismo. Por último, se mostrarán posibles espacios de mejorar y ampliaciones del trabajo que podrían realizarse en un futuro.

2

Estado del arte

El crecimiento y desarrollo del mundo tecnológico donde vivimos no deja de aumentar, por lo que debemos estar informados del conocimiento actual y las nuevas ideas y soluciones que van apareciendo. En este capítulo hablaremos sobre varios elementos clave de los nuevos entornos basados en SDN, como son algunos protocolos de programabilidad y la definición de modelos de datos estándar, las *White-Boxes*, y dos proyectos *OpenSource* que serán clave en el desarrollo de nuestro trabajo.

2.1. Redes definidas por software (SDN)

Las redes definidas por software (más comúnmente conocidas como *Software-Defined Networks (SDN)*) son un enfoque de la arquitectura de redes de comunicaciones, que aísla los recursos de red en un sistema virtualizado, es decir, el software se encuentra desvinculado del hardware. La SDN separa los dos planos de los dispositivos de red, ya que traslada al software el plano de control que establece dónde enviar el tráfico y deja en el hardware el plano de datos que realmente reenvía el tráfico. De este modo, los operadores de red podrán controlar los dispositivos de una forma centralizada, sin preocuparse de la compleja topología que puedan tener, y sin tener que administrar cada uno de forma manual [11].

Las SDN proporcionan ciertas ventajas sobre las redes tradicionales:

- **Control y flexibilidad**: Las SDN permiten que los operadores puedan controlar distintos dispositivos al mismo tiempo, sin estar condicionados por las restricciones del fabricante de cada uno de ellos. No solo importa su facilidad, sino también la flexibilidad que ahora existe al tener una virtualización de la red, ya que se pueden aumentar o reducir los recursos de red cuando se considere oportuno, sin necesidad de tener que agregar otra pieza de hardware del mismo fabricante.
- **Abstracción de la red**: Los servicios y aplicaciones basados en tecnologías SDN siguen un enfoque desagregado, en el que ya no existirá esa dependencia del fabricante del hardware. Ahora los operadores tendrán la posibilidad de elegir protocolos de código abierto para comunicarse con los distintos dispositivos de la red, por medio de APIs (*Application Programming Interfaces*), en lugar de siguiendo interfaces de configuración muy ligadas al hardware del fabricante.
- **Seguridad sólida**: En una SDN, la visibilidad de la red es mucho mayor, por lo que se facilita la detección de amenazas de seguridad y vulnerabilidades. Dada la proliferación de dispositivos inteligentes que se conectan

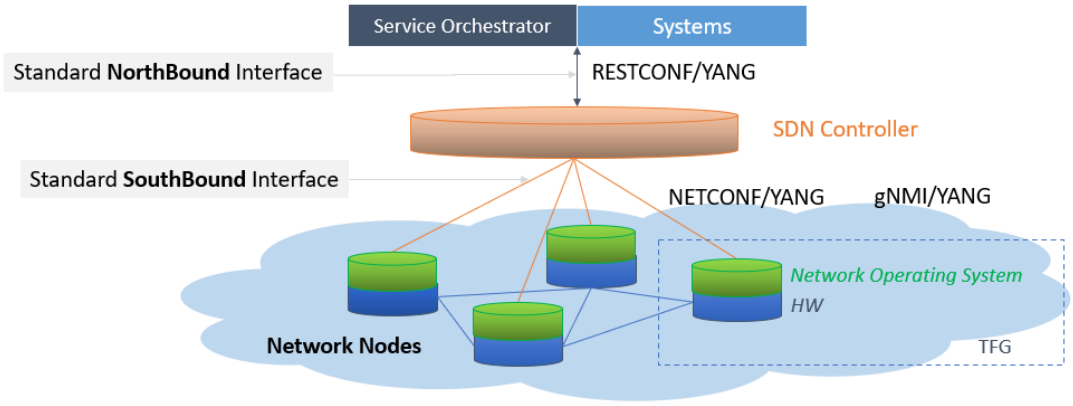


Figura 2.1: Ejemplo de entorno basado en SDN [14].

a Internet, una SDN ofrece ventajas patentes en comparación con las redes tradicionales. Los operadores podrán controlar la seguridad de cada dispositivo de un modo independiente e inteligente según sus necesidades, o poner en cuarentena rápidamente a cualquier dispositivo que pueda ser un riesgo para la red (por alguna vulnerabilidad o haber sido ya infectado), evitando que la amenaza se propague por ella.

- **Reducciones en el coste total de propiedad *(Total Cost of Ownership (TCO))*** : Las infraestructuras de SDN suelen tener un menor coste que sus equivalentes en hardware, ya que se ejecutan en servidores comerciales y no en dispositivos costosos de un solo uso. Además, al necesitar menos hardware físico, se logra la consolidación de los recursos, lo cual da como resultado una reducción de los costes generales, energéticos y de espacio físico.

Para la programabilidad de estos dispositivos de red basados en SDN, existen distintos protocolos. Vamos a mostrar algunos de ellos.

2.1.1. Protocolos

Los protocolos de gestión de redes que vamos a tratar son NETCONF y RESTCONF, pero como ambos comparten el mismo lenguaje de modelado de datos, comencemos por él. ¿Pero a qué nos referimos con lenguaje de modelado de datos? Haciendo el símil con la comunicación habitual entre personas, si los protocolos son el modo de combinar las palabras para formar frases y oraciones, el lenguaje de modelado de datos sería el idioma de esas palabras. Este lenguaje que comparten los protocolos anteriores es YANG, desarrollado por el IETF y publicado como RFC 6020 en octubre de 2010.

YANG puede usarse para modelar datos de configuración, datos del estado de los dispositivos de la red o el formato de las notificaciones de eventos emitidos en la red. También permite modelar la firma de las llamadas a procedimientos remotos (*Remote Procedure Call (RPC)*).

El lenguaje es modular y las estructuras de datos que representa siguen un formato de árbol XML.

Los modelos de datos YANG usan Xpath para definir los límites en los elementos del modelo. Más información sobre YANG se encuentra en mayor detalle en [4]. Pasemos a hablar de los dos protocolos mencionados, pero como YANG se creó para ser utilizado por NETCONF, comencemos por él.

El protocolo NETCONF (*Network configuration*) es un protocolo de gestión de redes desarrollado y estandarizado por el IETF, publicado como RFC 4741 en 2006, y luego revisado y publicado de nuevo en 2011 en el RFC 6241. NETCONF proporciona mecanismos para instalar, manipular y eliminar la configuración de dispositivos de red. Como ya hemos dicho, entiende el lenguaje YANG.

Podemos separar conceptualmente este protocolo en cuatro capas:

1.– La capa de Contenido, que consta de datos de configuración y notificación, que son intercambiados entre cliente y servidor.

2.– La capa de Operaciones, que define un conjunto base de instrucciones para obtener y editar los datos de configuración.

3.– La capa de Mensajes, que proporciona una vía para definir RPCs y notificaciones.

4.– La capa de Transporte, que permite la existencia de una comunicación segura y confiable de los mensajes enviados entre cliente y servidor.

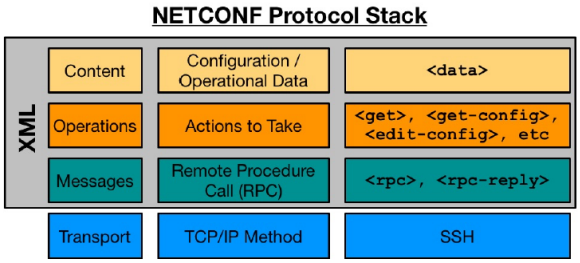


Figura 2.2: Pila del protocolo NETCONF [12].

Las mostramos en la figura 2.2, y más información sobre el protocolo NETCONF puede encontrarse en [6]. Pasemos ahora al siguiente protocolo.

RESTCONF presenta muchas similitudes con NETCONF, pero su mayor diferencia es el empleo de peticiones HTTP (o HTTPS) en lugar de RPCs. Se definió por el IETF en el RFC 8040 en 2017, y proporciona también una interfaz programática de acceso a datos modelados en lenguaje YANG. Estas son las capas de la pila del protocolo RESTCONF:

1.– La capa de Contenido, que es como la de NETCONF, pero añadiendo que los datos puedan tener formato JSON además de XML.

2.– La capa de Operaciones, que define las instrucciones en formato CRUD (*Create, Read, Update, Delete*), tal y como si estuviéramos en un API-REST.

3.– La capa de Transporte, que emplea HTTP o HTTPS como protocolo de transporte para la comunicación entre cliente y servidor.

Podemos verlas en la figura 2.3, y para encontrar más información sobre este protocolo, podemos visitar [3].

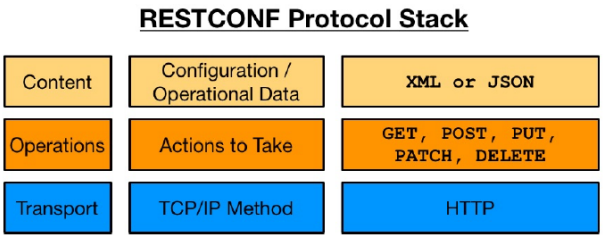


Figura 2.3: Pila del protocolo RESTCONF [12].

Hemos visto dos protocolos de gestión de dispositivos de red que entienden modelos descritos con YANG, pero dichos modelos pueden cambiar entre fabricantes. Para poder unificar esto, surge *OpenConfig*.

2.1.2. OpenConfig

OpenConfig es un esfuerzo colaborativo que busca llegar a un entorno de redes más dinámico y programable, basado en principios de las SDN, como la configuración declarativa y la gestión basada en modelos (*Model-Driven Management*), que puedan diseñar los propios usuarios [17].

Para ello, juntan un conjunto consistente de modelos de datos (escritos en YANG) que son independientes del fabricante. Estos modelos definen el estado configuracional y operacional de los dispositivos de red para protocolos y servicios de red comunes. El objetivo fundamental de *OpenConfig* es que, con un solo conjunto de modelos de datos, los operadores de red sean capaces de configurar y controlar todos los dispositivos de red que soporten la iniciativa *OpenConfig*.

Los modelos se encuentran en un repositorio público de GitHub en el que se puede colaborar, aportando nuevos modelos o ampliando los ya existentes. Cada vez más fabricantes tratan de que sus dispositivos soporten los modelos de *OpenConfig*, de modo que sigan la iniciativa de desagregación de la red que tanto trata de crecer en los últimos años.

2.2. *White-Boxes*

Podríamos definir básicamente el concepto de *White-Box* (caja blanca) como un entorno de red en el que el hardware soporta la instalación de cualquier software en él.

Por el momento hemos explicado una visión general de las SDN y su arquitectura, pero veamos la de las *White-Boxes*, donde podemos tener también una arquitectura modular. Los componentes hardware y software que formarán la “caja” (los mostramos en la figura 2.4) pueden ser elegidos separadamente por el proveedor.

- *Baremetal Hardware*: es hardware puro, formado por diferentes componentes (chips, memorias, *pluggables*...)

- *Network Operating System (NOS)*: Software instalado en el metal para proporcionar funcionalidades de red (L2, L3...).
- *Box-System*: consiste en juntar los dos anteriores.

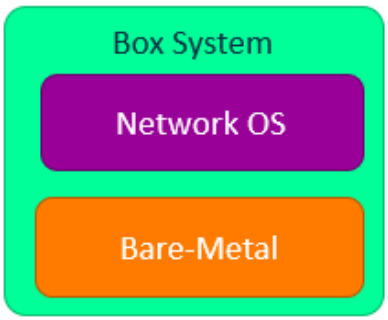


Figura 2.4: Elementos que forman una white box [15].

El hardware puro ha sido usado como parte de la infraestructura de centros de datos desde hace varios años. También se ha usado combinado con una red virtual controlada por un SDNc (*Software-Defined Network controller*) para ambas redes (física y virtual) [2].

Vamos a hablar ahora de los conmutadores de red o, como se nombran en inglés, *switches*. A partir de este momento nos referiremos a ellos de esta segunda manera, ya que es la más extendida.

Un switch es un dispositivo de interconexión utilizado para conectar equipos en red formando lo que se conoce como una red de área local (LAN) y cuyas especificaciones técnicas siguen el estándar conocido como Ethernet (o técnicamente IEEE 802.3). Por tanto, están en la capa de red de nivel 2 del modelo OSI (*Open Systems Interconnection*), aunque también existen algunos más recientes que integran funciones de nivel 3 propias de los *routers*.

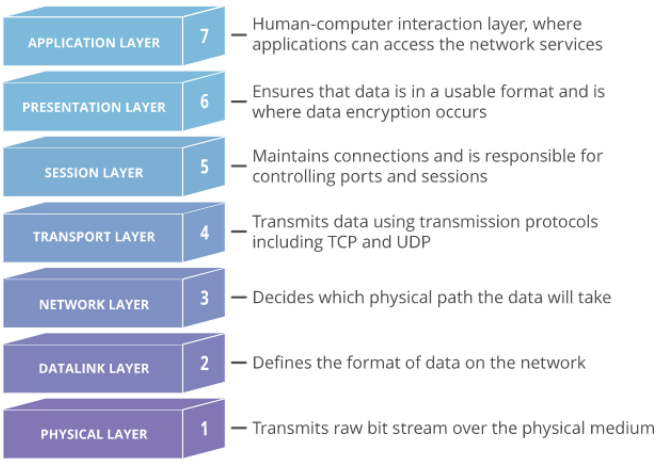


Figura 2.5: Capas del modelo OSI [5].

Estos dispositivos han sido hasta los últimos años muy poco tolerantes, ya que se componían de hardware y software del mismo fabricante. Sin embargo, recientemente han surgido las iniciativas de

SDN, y con ellas, los *White-box Switches* (o como los llamaremos, *White-Boxes*).

Antes de poder definirlos, vamos a mostrar la estructura de los componentes de un switch en la figura 2.6.

Figura 2.6: Arquitectura de un switch.

Un ASIC (circuito integrado para aplicaciones específicas) es un elemento específico de hardware destinado a una tarea concreta. En el caso de los switches, dicha tarea consiste en el envío de paquetes por la red de área local, como ya hemos mencionado. El hardware incluye varios componentes físicos como pueden ser puertos de entrada/salida, LEDs, etc. El NOS tiene la labor de controlar el ASIC para las tareas de red y de facilitar la comunicación entre el hardware y las aplicaciones del plano de control y administración. Por último, el plano de control y administración facilita al usuario herramientas para la gestión del switch y de su NOS.

En los switches tradicionales, tanto el hardware como el NOS eran propiedad del fabricante, por lo que se ofrecía ya dicho software instalado y no había modo de cambiarlo. En cambio, en las *White-Boxes*, se dota de libertad a los usuarios para instalar el NOS que desean en el hardware. Este nuevo ecosistema abierto crea nuevas oportunidades y retos en la industria de las telecomunicaciones, tal y como pasó en las arquitecturas de centros de datos. Ahora las soluciones desagregadas son cada vez más posibles, y eso recae en más ventajas para los operadores de red y los usuarios.

Pero esto no es trivial, ya que los switches no son dispositivos estandarizados. Esto quiere decir que cada diseño es diferente, por lo que conseguir que sean una *White-Box*, es decir, que soporten la instalación de cualquier NOS, no es tarea sencilla. Para que el hardware de un switch funcione correctamente junto con un NOS, se debe disponer de drivers de los componentes internos como sensores o LEDs, además del mapeo de los puertos físicos del ASIC con los del NOS.

Por tanto, esta transición de los switches tradicionales a las *White-Boxes* ha sido progresiva. En un primer lugar, se consiguió sustituir el ASIC propietario por piezas de silicio de distintos fabricantes, pero manteniendo hardware y software del mismo fabricante. Finalmente, se logró desagregar estos últimos dos elementos, apareciendo los conceptos de *OpenSoftware* (aquel que puede ser instalado en cualquier hardware) y *OpenHardware* (aquel que soporta la instalación de cualquier software).

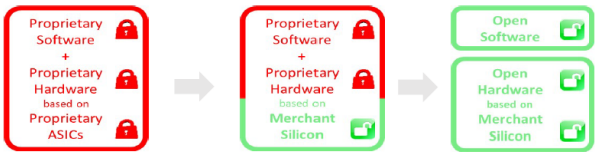


Figura 2.7: Evolución de la arquitectura interna de un switch [10].

Para el desarrollo de estas tecnologías existen muchos proyectos *OpenSource*, como los dos siguientes.

2.3. Proyectos *OpenSource*

Los proyectos *OpenSource* (proyectos de código abierto) están cada vez más presentes en nuestra comunidad y resultan realmente útiles a la hora de impulsar nuevas tecnologías. En este entorno basado en SDN y *White-Boxes*, existen distintos proyectos de código abierto.

Algunos proyectos de código abierto han dado lugar a software gratuito y totalmente libre de gran importancia mundial. Existen ejemplos como TensorFlow, MongoDB, Docker o Kubernetes (de estos dos últimos hablaremos en el próximo capítulo), pero posiblemente su mayor exponente haya sido Linux, que ha dado lugar al sistema operativo que más libertad proporciona a los usuarios. Además, ha contribuido en gran medida al desarrollo del sistema operativo móvil Android.

Aquí vamos a hablar de dos proyectos *OpenSource* relacionados con las redes que serán de gran importancia para el desarrollo de este TFG: SONiC y FRINX UniConfig.

2.3.1. SONiC

SONiC (*Software for Open Networking in the Cloud*) es un NOS *OpenSource* popular en la industria del *Open-Networking*. Es un NOS 100 % funcional, ya que contiene componentes para controlar el Ethernet ASIC y las interfaces de usuario. Está basado en Linux, pero es desarrollado por Microsoft.

Para soportar distintos Ethernet ASICs de los fabricantes, SONiC emplea SAI (*Switch Abstraction Interface*, otro proyecto *OpenSource*) internamente. De este modo, es compatible con todos los ASICs que tienen soporte para SAI.

Por otro lado, SONiC posee el SwSS (*Switch State Service*), que permite controlar el ASIC usando una base de datos central como interfaz entre las aplicaciones de red que corren en el switch y su propio hardware. La base de datos es Redis, y almacena toda las estadísticas, configuraciones de usuario e información relacionada con los Ethernet ASICs. Como hemos dicho, la mayor parte de la comunicación entre los distintos componentes de SONiC se realiza a través de Redis, en lugar de

directamente entre ellos. Esto permite que sea sencillo añadir una nueva funcionalidad a SONiC, por medio de, por ejemplo, *daemons* suscritos a algunos eventos determinados de Redis.

Por tanto, el SwSS dota a SONiC de las características de los entornos de *White-Boxes*, ya que permite a las aplicaciones de red ser ejecutadas sobre SONiC de una forma independiente del hardware.

Mostramos esta arquitectura en la figura 2.8.

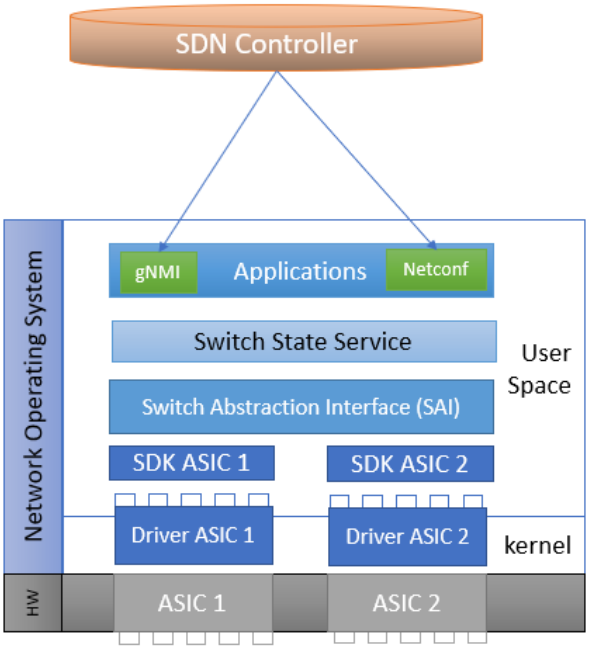


Figura 2.8: Arquitectura de SONiC.

2.3.2. FRINX UniConfig

FRINX UniConfig es una herramienta cuyo objetivo es gestionar el estado configuracional y recuperar el estado operacional de dispositivos de red físicos y virtuales. En futuras apariciones lo llamaremos UniConfig. Es un proyecto de código abierto desarrollado por FRINX [7], compañía que ofrece soluciones basadas en tecnologías *OpenSource* para automatizar procesos en redes y *clouds*. FRINX tiene sedes en Bratislava (Eslovaquia) y Nueva York (Estados Unidos), y algunas empresas que utilizan sus soluciones son Facebook o Vodafone.

UniConfig proporciona una única API para distintos dispositivos de la red. Puede ser ejecutado como una aplicación en hardware puro, en un entorno virtualizado (como una máquina virtual o un contenedor) o formando parte de FRINX Machine (otra herramienta automatizada de la compañía). UniConfig también posee un almacenamiento de datos que puede ejecutarse en memoria interna o en una base de datos externa.

La arquitectura de UniConfig puede verse en la figura 2.9:

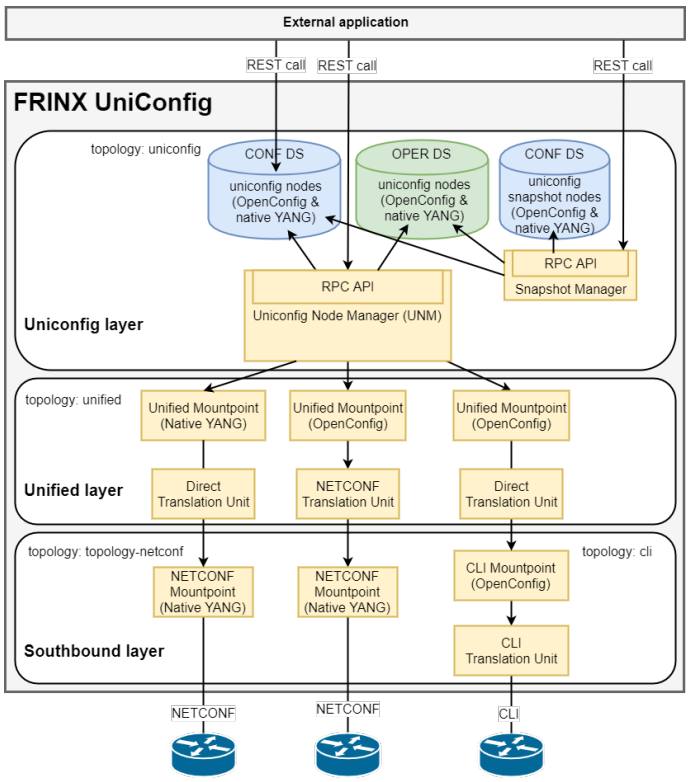


Figura 2.9: Arquitectura de UniConfig [9].

UniConfig consta de tres capas principales, cada cual proporciona un nivel mayor de abstracción de los elementos de red:

- La capa de UniConfig: posee el *UniConfig Node Manager (UNM)*, encargado de mantener la configuración de los dispositivos basados en una configuración intencionada, por medio de RPCs y un sistema de almacenamiento (*datastore*).
- La capa unificada: posee el *Unified Mountpoint*, que unifica la API para varios protocolos *Southbound* como NETCONF y CLI (*Command-Line Interface*, o interfaz de línea de comandos). Esta API se describe usando modelos de *OpenConfig* y emplea las unidades de traducción (*translation units*) para traducir entre datos *OpenConfig* y datos del punto de montaje *Southbound*.
- La capa *Southbound*: posee los puntos de montaje NETCONF y CLI, y unidades de traducción.

El sistema de almacenamiento se compone de dos bases de datos:

- Base de datos configuracional (*CONF DS*): Contiene el estado configuracional (las configuraciones que se quieren aplicar en los dispositivos). Las aplicaciones externas tienen permisos de lectura y escritura.
- Base de datos operacional (*OPER DS*): Contiene el estado operacional (las configuraciones que están corriendo actualmente en los dispositivos). Las aplicaciones externas tienen permiso solo de lectura.

Dentro de las unidades de traducción, nos interesarán más las de CLI (*CLI Translation Units*), pues ahí realizaremos la contribución. Una unidad de traducción CLI define un mapeo entre los datos CLI específicos de un dispositivo y modelos estándar de *OpenConfig* (escritos en YANG). La unidad de

traducción puede leer y escribir configuraciones o leer el estado de un dispositivo. Se comunica con el CLI del dispositivo mediante SSH o Telnet.

3

Diseño

En este capítulo se mostrarán las decisiones de diseño que se han tomado durante el desarrollo del trabajo. Primero, se hablará de la contenerización, un concepto que será importante a la hora de los dos diseños posteriores. Después, se explicarán los diseños del TFG: primero el basado en Goldstone y después el basado en SONiC, incluida su ampliación final.

3.1. Contenerización: Kubernetes y Docker

La manera tradicional de desplegar aplicaciones era instalarlas en un servidor usando el administrador de paquetes del sistema operativo. La desventaja era que los ejecutables, la configuración, las librerías y el ciclo de vida de todos estos componentes se entretejían unos con otros. Podíamos construir imágenes de máquina virtual inmutables para tener *rollouts* y *rollbacks* predecibles, pero las máquinas virtuales son pesadas y poco portables.

Para mejorar esto, surge la contenerización, que consiste en desplegar contenedores basados en virtualización a nivel del sistema operativo, en vez del hardware. Estos contenedores están aislados entre ellos y con el servidor anfitrión: tienen sus propios sistemas de archivos, no ven los procesos de los demás y el uso de recursos puede ser limitado. Son más fáciles de construir que una máquina virtual y, porque no están acoplados a la infraestructura y sistema de archivos del anfitrión, pueden llevarse entre nubes y distribuciones de sistema operativo.

Ya que los contenedores son pequeños y rápidos, una aplicación puede ser empaquetada en una imagen de contenedor. Esta relación uno a uno entre aplicación e imagen nos abre un abanico de beneficios para usar contenedores. Con contenedores, podemos crear imágenes inmutables al momento de la compilación en vez del despliegue ya que las aplicaciones no necesitan componerse junto al resto del stack ni atarse al entorno de infraestructura de producción. Generar una imagen de contenedor al momento de la compilación permite tener un entorno consistente que va desde desarrollo hasta producción. De igual forma, los contenedores son más transparentes que las máquinas virtuales y eso hace que el monitoreo y la administración sean más fáciles. Esto se aprecia más cuando los ciclos de vida de los contenedores son administrados por la infraestructura en vez de un proceso supervisor

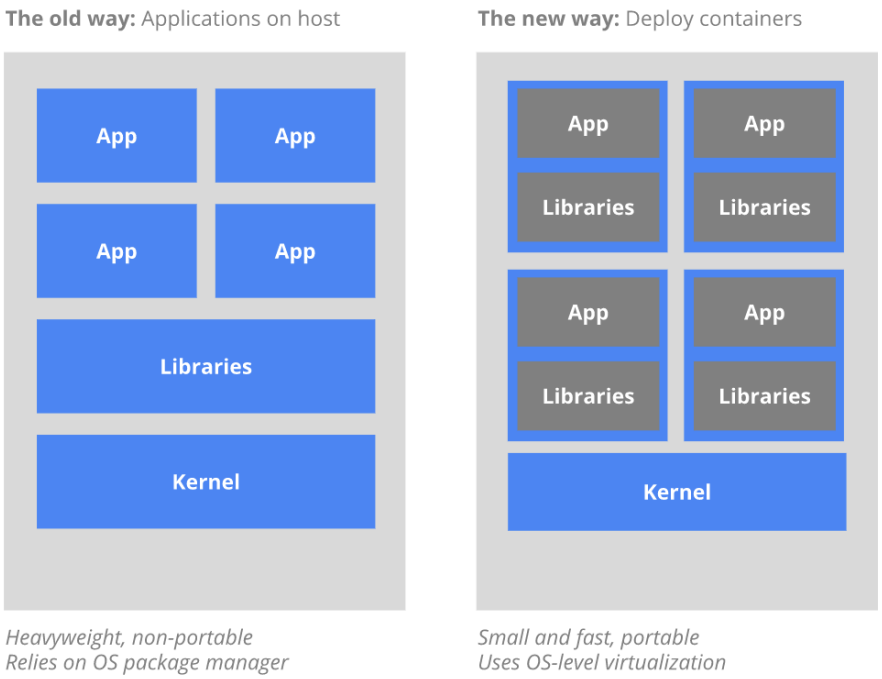


Figura 3.1: Arquitectura de aplicaciones desplegadas de modo tradicional (izquierda) y con contenerización (derecha) [13].

escondido en el contenedor. Por último, ya que solo hay una aplicación por contenedor, administrar el despliegue de la aplicación se reduce a administrar el contenedor.

Dos de sus exponentes más conocidos y empleados son los proyectos *OpenSource* Kubernetes y Docker.

3.1.1. Kubernetes

Kubernetes (*timonel* o *piloto* en griego), también conocido comúnmente como *K8s*, es una plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios, centrada en contenedores. Kubernetes facilita la automatización y la configuración declarativa. Ofrece la simplicidad de las Plataformas como Servicio (PaaS) con la flexibilidad de la Infraestructura como Servicio (IaaS), y permite la portabilidad entre proveedores de infraestructura.

Kubernetes fue diseñado inicialmente por Google, quien lo liberó en 2014, donándolo a la *Cloud Native Computing Foundation*. Está escrito en el lenguaje de programación *Go* y se puede desplegar en múltiples entornos *cloud* o en *bare-metal*. Soporta distintos *runtimes* de contenedores, como *containerd* o Docker. De este segundo vamos a hablar a continuación.

3.1.2. Docker

Docker (herramienta un nivel por debajo de Kubernetes) es un proyecto de código abierto creado por Solomon Hykes para automatizar la implementación de aplicaciones como contenedores. De manera similar a cómo una máquina virtual virtualiza (elimina la necesidad de administrar directamente) el hardware del servidor, los contenedores virtualizan el sistema operativo de un servidor. Docker se instala en cada servidor y proporciona comandos sencillos que puede utilizar para crear, iniciar o detener contenedores. Docker es también una empresa que promueve e impulsa esta tecnología, en colaboración con proveedores de la nube, Linux y Windows, incluido Microsoft.

Puede utilizar los contenedores de Docker como bloque de construcción principal a la hora de crear aplicaciones y plataformas modernas. Docker facilita la creación y la ejecución de arquitecturas de microservicios distribuidos, la implementación de código con canalizaciones de integración y entrega continuas estandarizadas, la creación de sistemas de procesamiento de datos altamente escalables y la creación de plataformas completamente administradas para sus desarrolladores.

La fácil creación y portabilidad de las aplicaciones usando Docker, permite que sea el entorno perfecto para el *testing* de las mismas.

3.2. Diseño del entorno virtual basado en Goldstone

El primer diseño del TFG del que vamos a hablar consistía en un entorno virtual basado en el sistema operativo Goldstone. Este es un NOS de código abierto para *Optical Packet Transponders*, del proyecto *Open Optical & Packet Transport* (OOPT). Utiliza distintos componentes *OpenSource* ya existentes, que han sido desarrollados en el *Open Compute Project* (OCP) y *Telecom Infra Project* (TIP), con el fin de ofrecer una solución *OpenSource* completa. Algunos de ellos son *Open Network Linux* (ONL), SONiC, SAI o *Transponder Abstraction Interface* (TAI).

En cuanto a la arquitectura de Goldstone, ONL se emplea como sistema operativo base y proporciona un amplio rango de soporte de dispositivos de *Open Networking*. Por encima de ONL se emplea Kubernetes para habilitar una gestión contenerizada de la aplicación, que se compone de una unión flexible y modular de software. SONiC/SAI se lanza como un grupo de contenedores cuando el hardware objetivo es un switch con Ethernet ASIC, mientras que TAI se usa cuando el hardware objetivo tiene componentes de transpondedor. Debido a su arquitectura modular, Goldstone podría ser extendido en un futuro para soportar dispositivos de red que no tengan Ethernet ASIC, pero incluyan transpondedores convencionales, ROADMs o amplificadores.

Goldstone se creó originalmente como un prototipo de NOS para el dispositivo Cassini, propuesto por NTT Electronics. Viene de los recientes avances en lograr una desagregación de la red, también para las redes ópticas. Para manejar estas redes lo que se emplean son transpondedores ópticos

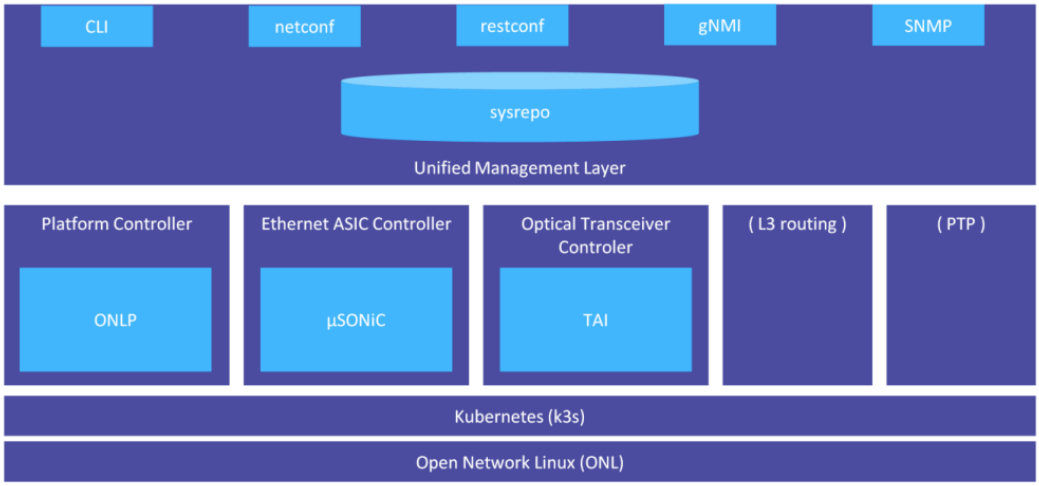


Figura 3.2: Arquitectura de Goldstone [16].

o, dispositivos más avanzados que combinan funciones tanto ópticas como de IP. Estos últimos se conocen como *Packet/Optical*, y ejemplos pueden ser Cassini o Galileo.

En nuestro entorno virtual con Goldstone, queríamos que se pudiera comprender su funcionamiento y realizar pruebas sobre su gestión. Para ello, se contaba con el soporte de Wataru Ishida (NTT Electronics), uno de los desarrolladores principales de Goldstone, y el repositorio *Goldstone Management Framework*.

Este entorno se compondría de contenedores gestionados por K3s, una versión más ligera de Kubernetes, ejecutados sobre un sistema virtualizado en primera instancia, con la intención de hacerlo en un dispositivo de red real habilitado en un futuro. Como sistema virtualizado primero se escogió una máquina virtual en el ordenador personal, pero debido a la cantidad de imágenes de Docker que se debían construir, se habilitó el acceso remoto a una máquina de uno de los laboratorios de Telefónica. Esta poseía gran memoria y velocidad de red, y permitía el acceso por VPN (se empleó *Cisco AnyConnect*). En ella se instaló un sistema operativo Ubuntu 18.04 sin interfaz gráfica, pero más que suficiente para realizar las primeras pruebas de creación y lanzamiento del Goldstone Management Framework.

Una vez comprobásemos que el entorno funcionaba correctamente, el siguiente paso sería implementar algún modelo de *OpenConfig* para TAI. Una vez implementado, trataríamos de configurar algún dispositivo óptico haciendo uso de él.

Sin embargo, este no fue el único NOS que probamos, pues también diseñamos un entorno virtual basado en SONiC, donde poder trabajar con él. De hecho, al cabo de unos meses se vio que la comunidad de SONiC era más grande, ofrecía más posibilidades de contribución a proyectos de código abierto, y permitía una más fácil integración del *driver* que queríamos desarrollar, por lo que decidimos seguir avanzando en dicho diseño y dejar Goldstone a un lado.

3.3. Diseño de la ampliación del entorno virtual basado en SONiC

El otro sistema operativo para el que diseñamos un entorno virtual fue SONiC. En primera instancia, reutilizaríamos el diseño del TFG de Javier Galán [10], sin usar la parte de telemetría. Este utiliza una topología de cuatro elementos: dos hosts (Host1 y Host2) y dos switches (Switch1 y Switch2). Todos serán lanzados en contenedores con Docker y tendremos Ubuntu 14.04 en los hosts y *SONiC P4 Software Switch* en los switches [1]. Se establecería una comunicación entre los hosts a través de los switches. Aquí dejamos de utilizar la máquina remota y volvimos a utilizar la máquina personal, ya que permitía más control y esta topología no requería tanta memoria.

Como hemos mencionado en la sección anterior, probamos ambos sistemas operativos y decidimos continuar el desarrollo del diseño de SONiC, ya que poseía una comunidad más amplia que ofrecía más posibilidades. Además, el proyecto de FRINX UniConfig resultó atractivo de cara a la contribución que se quería realizar.

Dicha contribución consistiría en la implementación de una unidad de traducción de CLI para SONiC. Con ella, podríamos configurar un dispositivo en el que estuviera corriendo el sistema operativo SONiC, enviando peticiones que siguen un modelo *OpenConfig*, pero que serán traducidas al CLI de SONiC. Como ya teníamos unos contenedores usando este sistema operativo, podríamos usarlos para probar la unidad de traducción y, en un futuro, podría probarse en un dispositivo físico.

Estas unidades de traducción pueden ser para distintos elementos de red, así que durante esta implementación la única decisión de diseño tomada fue la de qué unidades se iban a implementar. La *Init unit* era obligatoria, pues permitía *montar* el dispositivo. Las otras unidades que se implementaron fueron la *Configuration Metadata unit*, necesaria para ver si el estado configuracional y operacional del dispositivo coinciden (es decir, que el dispositivo está *in-sync*), y la *Interface unit*, necesaria para las operaciones sobre las interfaces del dispositivo.

Una vez terminamos la unidad, para poder comenzar a probarla, tuvimos que realizar ampliaciones y modificaciones del diseño anterior basado en SONiC.

El primer paso fue que, dado que UniConfig se comunica con el dispositivo mediante SSH, tuvimos que habilitar la conexión por este puerto en los switches, modificando el script *start.sh* para crear un único contenedor con SONiC, accesible mediante SSH.

El siguiente paso fue diseñar una topología donde pudiéramos probar la unidad de traducción de CLI, en tiempo real. Esta descartaría los hosts y se quedaría con los dos switches, pero esta vez accesibles mediante SSH. Entre ellos estableceríamos una comunicación directa por medio de *pings* de Switch2 a Switch1, a través de una interfaz (por ejemplo Ethernet1). Mientras tanto, utilizaríamos UniConfig para, por medio de la unidad de traducción implementada, realizar operaciones que afectasen

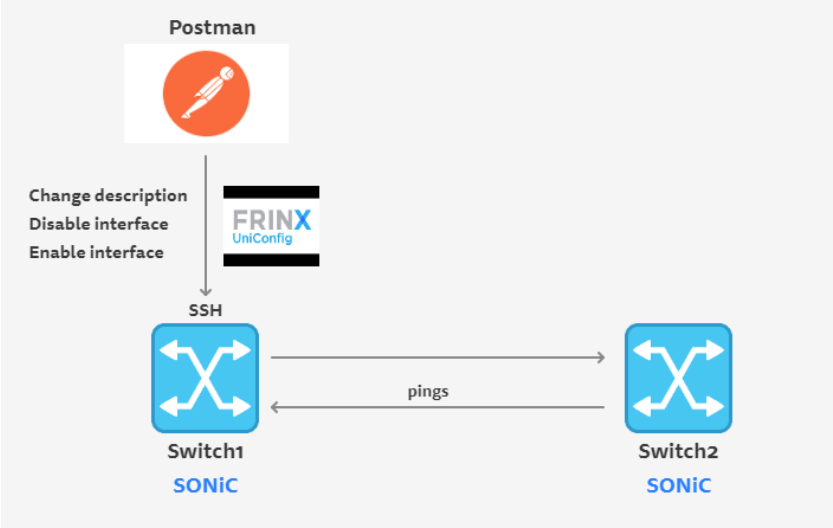


Figura 3.3: Topología final del entorno basado en SONiC.

o no a la comunicación. Lo primero sería cambiar la descripción de dicha interfaz, lo cual no debería alterar la conectividad entre los switches, pero lo siguiente sería deshabilitar tal interfaz. Tras esto, deberíamos ver cómo se pierde la conexión. Por último, se volvería a habilitar la interfaz y volveríamos a ver restablecida la comunicación.

4

Desarrollo

En este capítulo vamos a explicar el desarrollo del proyecto. Primero, hablaremos de la configuración de Goldstone desde el repositorio de código abierto sobre el que trabajábamos, y de los scripts implementados para lanzar el entorno. Después, narraremos el desarrollo de la unidad de traducción de SONiC, implementada como contribución a FRINX UniConfig. Por último, vamos a ver los pasos realizados a la hora de ampliar el entorno virtual basado en SONiC con la topología antes mencionada, en el que se podrían realizar las pruebas.

4.1. Configuración de Goldstone

Durante la primera parte del proyecto, cuando se trabajaba sobre Goldstone, se necesitaba tener un entorno estable con este NOS. Como ya dijimos antes, para ello íbamos a usar el código del *Goldstone Management Framework*.

En él se usa *sysrepo* como infraestructura central de configuración. Se basa en modelos de YANG, por lo que también se incluyen en el repositorio los propios modelos nativos YANG de Goldstone. De todos modos también existen *daemons* de traducción, que permiten el uso de modelos de YANG estándar como *OpenConfig* o IETF.

Este repositorio también incluye otros *daemons* para la gestión del NOS, que interactúan con sysrepo:

- North Daemon: Provee al usuario de una *northbound interface* (CLI, SNMP, NETCONF, RESTCONF, gNMI...). Existen *daemons* para los tres primeros actualmente.
- South Daemon: Actúa como enlace entre sysrepo y el controlador del hardware de la plataforma. Existen *daemons* para ONLP, SONiC/SAI y TAI actualmente.

Para lanzar el entorno en local (o como explicamos antes, en la máquina remota por VPN que terminamos utilizando), Wataru indicó que teníamos que seguir los pasos recogidos en el *deploy manifest* de su CI (*Continuous Integration*). Estos se encontraban en este fichero.

Se debía instalar K3s (un Kubernetes más ligero) y crear un cluster de Kubernetes de un único

nodo. Después habría que lanzar el siguiente manifiesto YAML de K8s a ese cluster creado anteriormente.

Si todo ha ido correctamente, deberían estar corriendo los *daemons* antes mencionados, y para comprobar su funcionamiento se miraría el de NETCONF, que usa un servidor Netopeer2. Ejecutando estas líneas del primer manifiesto mencionado, deberíamos ver una correcta comunicación.

Tras la labor de documentación sobre el uso de tecnologías de contenerización como K8s y K3s, realizamos las primeras pruebas de lanzamiento del entorno. Primero seguimos los pasos de un modo manual, lo cual no era un proceso corto (especialmente la parte donde se descargan y construyen las imágenes de Docker), y encontramos algunos errores a la hora final de comprobar con Netopeer2 que la comunicación en el entorno funcionaba.

Tras distintas pruebas e ideas, finalmente en una conversación con Wataru se encontró un *bug* en el manifiesto K8s, que estaba produciendo el error. Una vez solucionado, el entorno se logró lanzar correctamente. Durante este proceso, para no tener que realizar todas las pruebas a mano, se implementaron unos scripts para automatizar el proceso. Se dividió el proceso de lanzamiento en tres fases, cada una con un script independiente, y existía un primer script que exportaba la variable *K = sudo k3s kubectl*, que iba a ser de gran utilidad en el resto de los scripts, y posteriormente iba ejecutando cada uno de los otros tres scripts en el orden correcto. En la figura 4.1 mostramos parte del último, aunque en su totalidad se encuentran en nuestro repositorio de GitHub:

`https://github.com/Jandrov/SONiC/tree/master/material_extra/scripts_goldstone`

En este último script resultaron necesarias las dos líneas recuadradas en la figura 4.1, ya que con el comando *sed* se corrige el *bug* en el yaml afectado, y con *sudo k3s kubectl apply -f k8s/* se actualizan los contenedores para su correcto funcionamiento.

script3.sh

```
cd ~/environment/goldstone-mgmt/
$K exec ds/gs-mgmt-cli -- gscli -c "sonic; port; show" | grep Ethernet1

sed -i '16s/mgmt/mgmt-np2/' k8s/mgmt.yaml
$K apply -f k8s/

# A partir de aquí, ejecuto de las líneas 45 a 85 del https://github.com/microsonic/goldstone-mgmt/blob/master/.github/workflows/ci.yaml
export NP2_POD=$($K get pod -l app=gs-mgmt-np2 -o=jsonpath='{.items[0].metadata.name}{"\n"}')
export NP2_ADDR=$($K get svc netopeer2 -o=jsonpath='{.spec.clusterIP}{"\n"}')

ssh-keygen -f id_rsa -N ""
$K exec $NP2_POD -c netopeer2 -- mkdir -p /root/.ssh
$K cp id_rsa.pub $NP2_POD:/root/.ssh/authorized_keys -c netopeer2
cat <<EOF > config.xml
<modules xmlns="http://goldstone.net/yang/tai">
<module>
  <name>0</name>
  <config>
    <name>0</name>
    <admin-status>down</admin-status>
  </config>
</module>
</modules>
```

Figura 4.1: Parte del código de los scripts de instalación del *Goldstone Management Framework*. Comienzo del último script, donde se enmarcan las líneas que corrigen el *bug* encontrado.

Se trabajó con este repositorio en los meses de noviembre y diciembre, pero después ha sufrido bastantes cambios por una *major update* en el mes de abril, por lo que puede que estos scripts no se encuentren en completa sincronía con el estado actual del repositorio. De hecho, el *bug* encontrado ya ha sido corregido en esa última actualización.

4.2. Desarrollo de la unidad de traducción de SONiC en FRINX

El propósito de la contribución a UniConfig era desarrollar una unidad de traducción de CLI (o como diremos durante el resto del capítulo, *cli-unit*) para el sistema operativo SONiC, pues no existía en su repositorio.

Para obtener un primer conocimiento sobre FRINX y en concreto, FRINX UniConfig, y tener soporte a la hora del desarrollo de la cli-unit, se mantuvo contacto con Šimon Mišenčík, estudiante que trabaja como Software Engineer en FRINX.

Una vez se comprendió el funcionamiento de la plataforma, el primer paso fue preparar el entorno de desarrollo donde iba a realizarse la implementación y posteriores pruebas. Para ello se requerían:

- Postman: Para la comunicación con UniConfig.
- Java: Pues el código se encuentra escrito en este lenguaje.
- Maven: Para la gestión del proyecto.
- IntelliJ IDEA (opcional, pero útil para trabajar con código Java): Para el desarrollo del código y la ejecución y depuración de los tests.

Obtuvimos acceso al repositorio privado de FRINX, en Gerrit, del que clonamos dos repositorios, los cuales probamos que podíamos construir con Maven:

- cli-units: Repositorio donde se encuentran implementadas las distintas unidades de traducción CLI y en el que vamos a añadir la nuestra de SONiC.
- distribution: En este no se va a necesitar modificar nada, pero es necesario para las pruebas en local, ya que contiene el código del servidor de UniConfig, que recibirá las peticiones desde Postman y empleará la cli-unit apropiada para la gestión del dispositivo de red que queremos.

Con todo listo, el siguiente paso fue lanzar los contenedores con SONiC de nuestro entorno virtual, y conectándonos a ellos, emplear el comando *vtysh* para abrir una *shell* de Quagga en él. Desde ahí, se exploraron los comandos disponibles, como son:

- *show running-config*: Obtiene la configuración actual del dispositivo.
- *configure terminal*: Habilita el modo de configuración del dispositivo, cambiando el prompt para indicarlo. Después con *interface <nombre_interfaz>* se puede acceder a configurar una de ellas con comandos como:
    - *shutdown / no shutdown*: Habilita o deshabilita una interfaz, respectivamente.

- *description <nueva_descripcion>/ no description*: Cambia la descripción de la interfaz por <nueva_descripcion>o elimina la descripción actual, respectivamente.

Estudiar cómo funcionan estos comandos de CLI directamente sobre el dispositivo será crucial luego a la hora de implementar nuestras unidades.

Ya hecho esto, creamos el commit en Gerrit sobre el que vamos a trabajar, ya que el modo de añadir nuevos cambios es usando *git commit −−amend* sobre un commit inicial previamente firmado.

El proyecto cli-units se compone de un directorio por cada sistema operativo para el que se implementa una unidad de traducción, además de uno general *artifacts*, en el que se listan todos los módulos, unidades y futuros de cada módulo. Cada uno contiene ficheros *pom.xml* que va a emplear Maven a la hora de construir el proyecto. Por tanto, para el desarrollo de nuestra cli-unit de SONiC, tuvimos que modificar y crear varios de estos ficheros pom.xml.

Por otro lado, dentro de los directorios de cada NOS, existen otros subdirectorios referentes a cada unidad implementada. Hay tres unidades básicas:

- init: Se encarga de asegurar la inicialización con el dispositivo, configurando el *CLI prompt* y definiendo una estrategia de *rollback*. Además, debe contener un fichero donde se indiquen los dispositivos que se van a soportar (por ejemplo, según distintas versiones del sistema operativo). En caso de ser todos, se indica la versión con un asterisco. No contiene ningún *reader* o *writer*.
- configuration-metadata: Se utiliza para obtener el estado operacional del dispositivo. Por tanto, es necesaria para ver si este está *in-sync*, es decir, que ambos estados operacional y configuracional coincidan. Contiene un *reader*, que emplea la salida del comando *show running-config* mencionado anteriormente.
- it: Se emplea para testear la inicialización con un dispositivo físico.

Por tanto, nuestro commit comenzó por estas unidades, aunque en la *it unit* solo creamos el pom.xml, debido a que no teníamos acceso todavía a un dispositivo físico con SONiC, por lo que se dejaría para un futuro en caso de ser posible.

Existen otras unidades posibles dependiendo de los modelos del NOS que nos interesen. En estas, existen dos tipos de manejadores (*handlers*):

- Readers: Son responsables de leer y *parsear* los datos que vienen de los dispositivos. Para ello, emplean expresiones regulares que coincidan con la salida del comando CLI que ejecutemos en los dispositivos. Deben implementar el método *readCurrentAttributes*.
- Writers: Responsables de realizar cambios en los dispositivos. Deben implementar los tres métodos para escribir (*writeCurrentAttributes*), actualizar (*updateCurrentAttributes*) y eliminar (*deleteCurrentAttributes*), con el fin de que exista un correcto procedimiento de *rollback* de la herramienta. Para realizar estas operaciones, emplean plantillas que acepten determinados valores y permitan la ejecución del comando de CLI correspondiente en el dispositivo. Para cada writer debe existir un reader correspondiente, y debe manejar los posibles errores que produzca el dispositivo, por medio de excepciones. Solo permiten alterar datos configuracionales, no operacionales.

Más información general sobre las cli-units se puede encontrar en [8].

La que decidimos implementar fue la *interface unit*, que nos permitiría realizar ciertas operaciones

sobre las interfaces del dispositivo. Hay un directorio *src* y otros dos en su interior: *main* y *test*. En el primero estará el código de la unidad y el de los manejadores, y en el segundo estarán los unit tests de cada uno de los manejadores.

Dentro de *main* tenemos el fichero **SonicInterfaceUnit.java**, el cual es la base de nuestra unidad de traducción, ya que se encarga de definir su nombre, las versiones de SONiC soportadas, los esquemas YANG a utilizar (usaremos los de *OpenConfig* que se encuentran en el propio repositorio de FRINX, los cuales poseen algunas extensiones además de los modelos habituales), y los manejadores (readers y writers) que vamos proporcionar.

Después, en el directorio *handler*, se encuentran dos readers y un writer. Para los primeros hemos tenido que implementar los patrones que coinciden con ciertas partes que nos interesan de la salida del comando *show running-config*. Para ello, ha sido de gran utilidad la web regex101:

- **InterfaceReader.java**: Permite listar las interfaces del dispositivo. La expresión regular que usa se muestra en 4.2.
- **InterfaceConfigReader.java**: Permite leer la configuración de una interfaz del dispositivo. Sus expresiones regulares se muestran en 4.3.
- **InterfaceConfigWriter.java**: Permite alterar la configuración de una interfaz del dispositivo. Las plantillas implementadas se muestran en 4.4.

```
private static final String SH_INTERFACE_CFG = "show running-config";

private static final Pattern INTERFACE_ID = Pattern.compile("(interface )(?<id>\\S+)");
```

Figura 4.2: Parte del código de *InterfaceReader.java*.

```
private static final String SH_INTERFACE_CFG = "show running-config";
private static final String SINGLE_INTERFACE_STR = "(interface %s)(?s).*?(interface %s|^!|^router)";
private static final String LAST_SINGLE_INTERFACE_STR = "(interface %s)(?s).*";

private static final Pattern INTERFACE_ID = Pattern.compile("^interface (?<id>\\S+)$");
private static final Pattern INTERFACE_DESCRIPTION = Pattern.compile("^description (?<desc>.*)");
```

Figura 4.3: Parte del código de *InterfaceConfigReader.java*.

```
private static final String WRITE_TEMPLATE_SONIC = "configure terminal\n"
        + "interface {$data.name}\n"
        + "{% if ($enabled) %}shutdown\n"
        + "{% else %}no shutdown\n{% endif %}"
        + "{% if ($data.description) %}description {$data.description}\n"
        + "{% else %}no description\n{% endif %}"
        + "end\n";

private static final String DELETE_TEMPLATE_SONIC = "configure terminal\n"
        + "interface {$data.name}\n"
        + "no shutdown\n"
        + "end\n";
```

Figura 4.4: Parte del código de *InterfaceConfigWriter.java*.

Estos manejadores también poseen ciertos métodos para el parseo y navegación dentro de la salida del comando recibida.

Para probar todos estos manejadores, en *test* se encuentran unit tests de cada uno de ellos. Los de los readers se componen de unas cadenas de texto al comienzo con un ejemplo de salida del comando *show running-config*, una lista o cadena de texto con el resultado esperado, y un método para comprobar por medio de *asserts*, que el resultado de la ejecución del test coincide con el esperado. Para el writer comenzamos con ejemplos de plantillas de entrada según cada acción, un método para configurar e inicializar una interfaz para los tests, y finalmente los tests para cada una de las acciones, también empleando *asserts* sobre un resultado esperado.

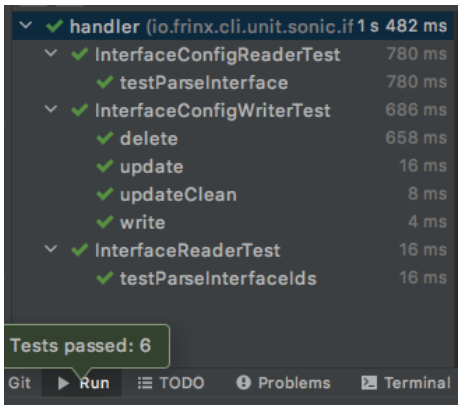


Figura 4.5: Muestra de que los tests implementados pasan correctamente.

Tras terminar la implementación de los tests y comprobar que estos funcionaban correctamente (figura 4.5), necesitamos probar el correcto funcionamiento de la cli-unit de SONiC, por lo que ampliamos el entorno virtual que teníamos originalmente, para obtener la topología descrita en la figura 3.3.

4.3. Ampliación del entorno virtual basado en SONiC

Para la construcción de nuestro propio entorno virtual basado en SONiC, comenzamos con un *fork* del repositorio SONiC de GitHub de Javier Galán (en el que usó la topología descrita en [1]), al que fuimos ampliando. Por tanto, todo el código que vamos a explicar a continuación se encuentra en nuestro repositorio público SONiC:

`https://github.com/Jandrov/SONiC`

El primer paso fue limpiar y actualizar un poco el repositorio, pero manteniendo todavía la topología anterior. Con ella, pudimos probar los comandos de CLI en SONiC, que luego empleamos para la implementación de la cli-unit.

Después, ya que en el estado actual del repositorio los contenedores de SONiC no pueden ser accesibles por SSH, tuvimos que investigar cómo lograrlo e implementamos un script para ello: **start_ssh_sonic.sh**. En él se siguen los siguientes pasos (requiriendo la ejecución de los scripts *install_requirements.sh* y *load_image.sh* al menos una vez antes, tal y como hacía falta para el entorno original de SONiC P4):

1.– Se obliga al uso del script del modo *source start_ssh_sonic.sh <root_password>*, indicando la contraseña del usuario *root* que usaremos para conectarnos. Se pide el uso de *source* para poder acceder al valor de la variable *SONIC_ADDR*, que se exporta posteriormente.

2.– Se crea y arranca el contenedor con SONiC, pero eliminando la flag *−−net=none* que se utilizaba en el entorno original. Esto es porque querremos que exista una IP para este contenedor.

3.– Se ejecuta el script */sonic/scripts/startup.sh* en el contenedor, para inicializar SONiC en él correctamente. Después se espera un minuto para que todo se configure correctamente, y se comprueba en la lista de contenedores activos.

4.– Se emplea el comando *chpasswd* para cambiar la contraseña del usuario root y acceder con ella luego por SSH. Después, se modifica el fichero */etc/ssh/sshd_config* con el comando *sed*, y finalmente se reinicia el servicio SSH con *service ssh restart*. Se duermen 10 segundos para que todo se configure bien.

5.– Por último, se obtiene la IP del contenedor y se exporta en *SONIC_ADDR* por medio de *docker inspect*.

```
#!/bin/bash

NAME=sonic-ssh

if [ ! $1 ]; then
    echo "[ERROR] Usage is: 'source start_ssh_sonic.sh <root_password>'"
    return
fi

sudo docker run --privileged --entrypoint /bin/bash --name $NAME -it -d -v $PWD/switch1:/sonic docker-sonic-p4:latest
sudo docker exec -d $NAME sh /sonic/scripts/startup.sh

sleep 60
sudo docker ps

sudo docker exec -d $NAME bash -c "echo 'root:$1' | chpasswd"
sudo docker exec -d $NAME sed -i 's/without-password/yes/' /etc/ssh/sshd_config
sudo docker exec -d $NAME service ssh restart

sleep 10

export SONIC_ADDR=$(sudo docker inspect -f "{{ .NetworkSettings.IPAddress }}" $NAME)
echo "Container $NAME created and running with $SONIC_ADDR IP address"

```

Figura 4.6: Código del primer script (en este commit) para lanzar contenedores de SONiC accesibles por SSH.

También implementamos otro para detener y eliminar estos contenedores accesibles por SSH, en el script **stop_ssh_sonic.sh**.

Tras esto, continuamos con la implementación de la cli-unit hasta llegar al punto donde debíamos probar su funcionamiento. Entonces diseñamos la topología de la figura 3.3 y para ella, hubo que actualizar el script start_ssh_sonic.sh.

La primera actualización fue la de permitir que funcionase en Mac correctamente, ya que se cambió el ordenador donde se realizaba el desarrollo de Linux a Mac, y Docker tiene diferencias entre esas plataformas. En Mac el contenedor se encontrará en *localhost* y necesitaremos asociar un puerto distinto al de SSH (que es el 22). Para comprobar que estamos en Mac, miramos que el valor del comando *uname* sea ”Darwin“.

El último paso fue el de modificar el script para lanzar dos contenedores similares, aunque expo-

```
11  KERNEL_NAME=$(uname)
12
13  # Docker on Mac requires mapping a different port to standard SSH port
14  if [[ "$KERNEL_NAME" == "Darwin" ]]; then
15      sudo docker run --privileged --entrypoint /bin/bash --name $NAME -it -d -p 2022:22 -v $PWD/switch1:/sonic docker-sonic-p4:latest
16  else
17      sudo docker run --privileged --entrypoint /bin/bash --name $NAME -it -d -v $PWD/switch1:/sonic docker-sonic-p4:latest
18  fi
```

Figura 4.7: Parte del código de start_ssh_sonic.sh (en este commit) donde se comprueba si el anfitrión es una máquina Mac.

niendo un puerto distinto como SSH para cada uno de ellos (ya que recordemos que ambos están en localhost). También limpiamos el código para extraer partes duplicadas y usamos bucles y mensajes para mostrar el progreso del script. Por último, el paso más importante fue el de crear nuestra propia red de Docker y conectar a los contenedores a dicha red. De ese modo, podrán comunicarse entre ellos por medio de pings enviados a la interfaz generada automáticamente por Docker al conectar cada contenedor a la red creada, con nombre *my-network*. La IP de esta interfaz, que es *eth1*, la obtenemos por medio del comando *grep* con una expresión regular para direcciones IPs, desde la salida del comando *docker network inspect*.

```
55  # Creamos nuestra propia red y conectamos los switches a dicha red
56  # We create our own network and connect the switches to that network
57  sudo docker network create $NETWORK
58  for name in "${NAMES[@]}"; do
59      echo "Adding $name to the network $NETWORK"
60      sudo docker network connect $NETWORK $name
61  done
62
63  # Obtenemos las direcciones IP asignadas a cada contenedor en nuestra red
64  # We obtain IP addresses from our network assigned to each container
65  ADDRESSES=($SWITCH1_ADDR $SWITCH2_ADDR)
66  PORTS=($SWITCH1_PORT $SWITCH2_PORT)
67  i=0
68  sudo docker network inspect -f "{{ .Containers }}" $NETWORK | grep -Eo '\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\/\d{1,2}' | \
69      while read ip; do \
70          echo "Container ${NAMES[$i]} created and running with ${ADDRESSES[$i]} IP external address, $ip internal address and SSH port ${PORTS[$i]}"; \
71          ((i++)); \
72      done
```

Figura 4.8: Parte del código de start_ssh_sonic.sh (en su versión final) donde creamos y manejamos nuestra red de Docker.

Obviamente, también modificamos el script **stop_ssh_sonic.ssh** para eliminar estos contenedores, y actualizamos el README del repositorio para ilustrar este entorno de pruebas del TFG.

Con estos últimos scripts, podían por tanto lanzarse dos contenedores de Docker con SONiC, accesibles por SSH (en distinta IP y puertos según el sistema operativo anfitrión), con la contraseña del superusuario (root) que nosotros elijamos al comienzo, y que pueden comunicarse entre ellos por sus interfaces eth1. Ya tenemos todo listo para las pruebas y los resultados.

5

Pruebas y resultados

En este capítulo vamos a mostrar las pruebas realizadas sobre nuestro nuevo entorno virtual basado en SONiC. Primero, veremos la conectividad que queríamos que existiera. Después, demostraremos que la unidad de traducción de CLI implementada para UniConfig funciona. Por último, mostraremos el resultado de dicha contribución realizada al proyecto *OpenSource* de FRINX UniConfig.

5.1. Demostración de la conectividad en el entorno virtual

Una vez finalizados los cambios en los nuevos scripts del repositorio de SONiC, ya podemos ejecutarlos para lanzar nuestro entorno virtual.

Siguiendo los pasos del README:

1.– Ejecutamos el comando: *./install_requirements.sh*. Con este script, instalamos Docker en caso de necesidad.

2.– Ejecutamos el comando: *./load_image.sh*. Con este script cargamos y construimos la imagen de SONiC-P4 que utilizaremos para los switches.

3.– Ejecutamos el comando *source start_ssh_sonic.sh mypassword*. Con este script comenzaremos dos contenedores con el SONiC del directorio switch1, accesibles por SSH. Sus IPs estarán en las variables de entorno *SWITCH1_ADDR* y *SWITCH2_ADDR*, sus puertos SSH en *SWITCH1_PORT* y *SWITCH2_PORT*, y el usuario será *root* (con la contraseña *mypassword*). También se habrá creado la red de Docker “my-network” y los contenedores se habrán conectado a ella. Ambos tendrán entonces la interfaz *eth1* con distintas IPs v4.

Una vez todo termina, podemos mostrar los valores de las variables exportadas anteriormente, y podemos comprobar en la aplicación de escritorio de Docker (*Docker Desktop*) que los contenedores están corriendo. Podemos ver todo esto en la figura 5.1.

Ahora desde distintas terminales vamos a conectarnos a ellos, primero por medio de *docker exec*, y después por medio de SSH (tal y como hará posteriormente UniConfig).

Una vez conectados, vamos a mostrar las interfaces, en concreto *eth1*, pues es la que emplearemos en la prueba de conectividad entre los dos switches. Esto aparece en la figura 5.2.

El siguiente paso será hacer ping entre ambos switches. Como tenemos dos consolas abiertas des-

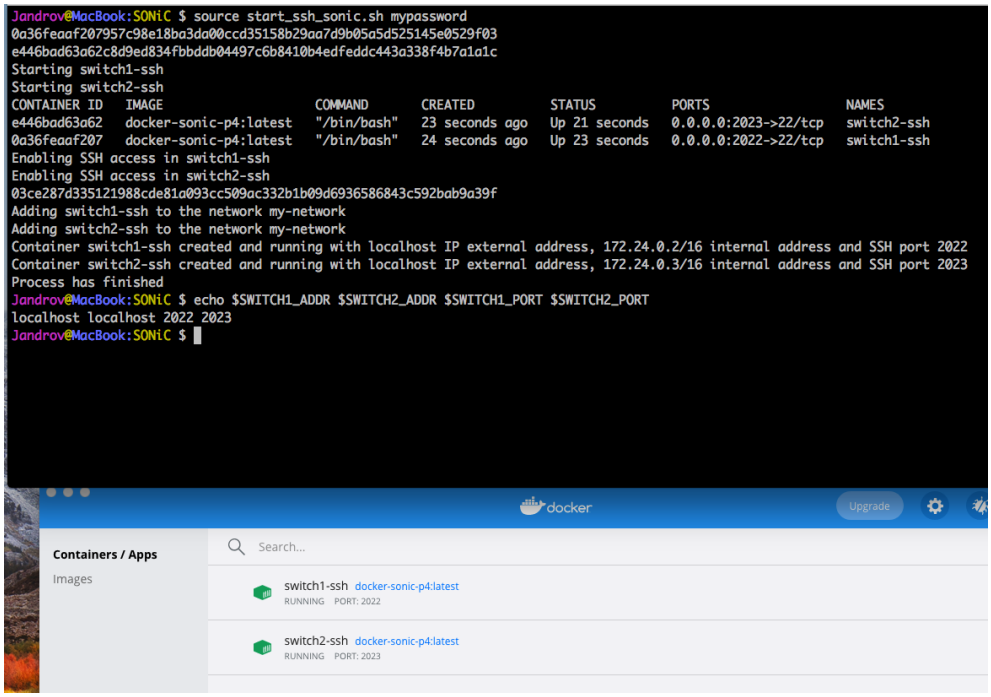


Figura 5.1: Ejecución del script start_ssh_sonic.sh.

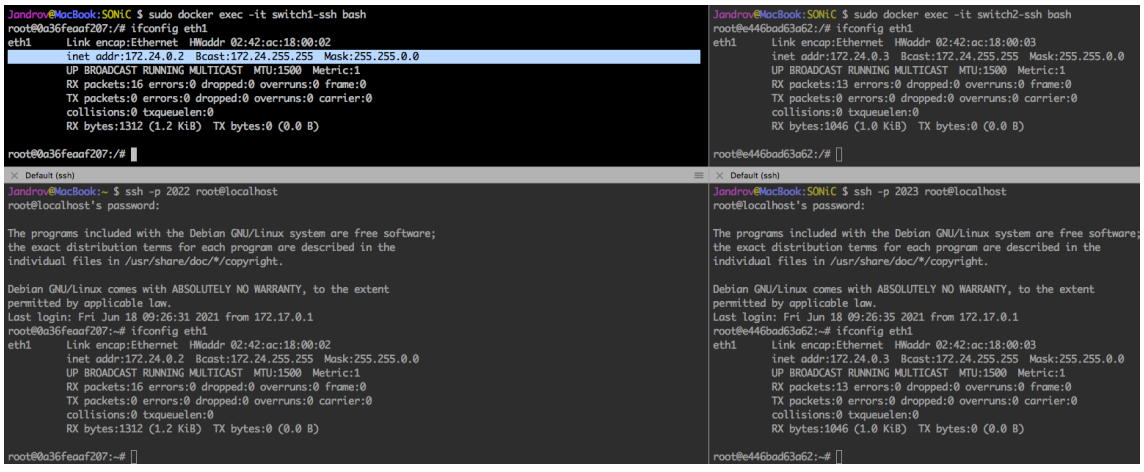
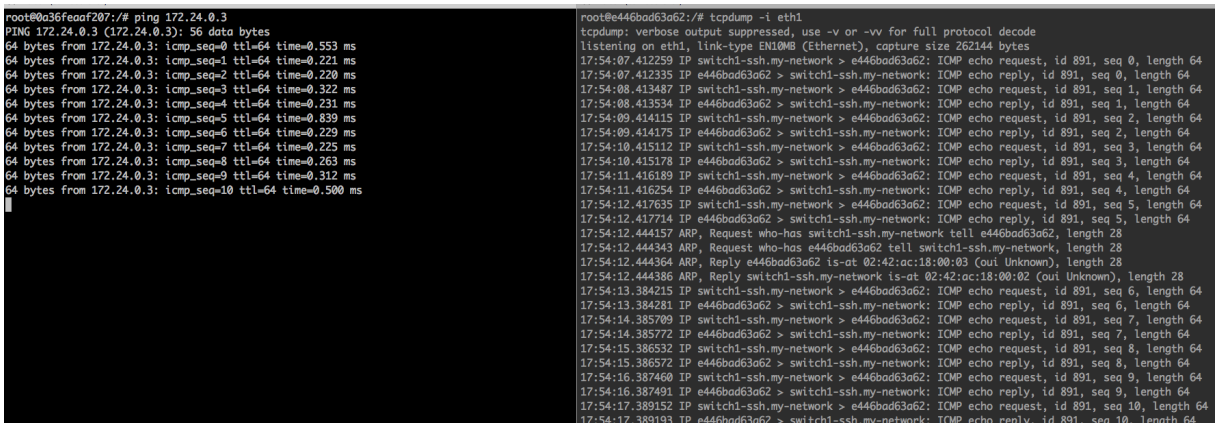
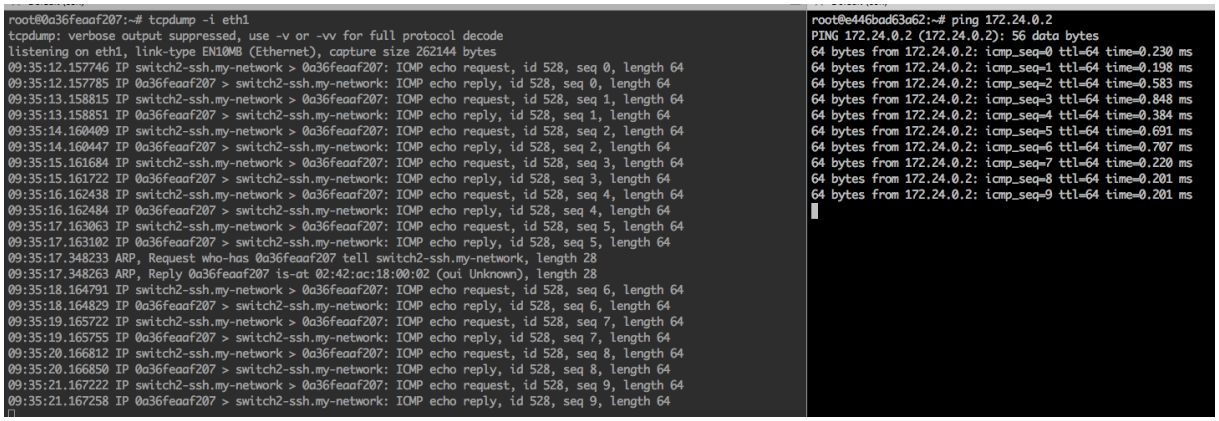


Figura 5.2: Comprobamos las IPs de los dispositivos en la interfaz *eth1* y vemos que coinciden con las impresas en la ejecución del script anterior.

de cada terminal, vamos a ejecutar *tcpdump -i eth1* en ambos switches, para mostrar más claramente la recepción de los paquetes ICMP del ping del otro switch. Podemos ver el resultado en la figura 5.3.



(a) Del switch1 al switch2.



(b) Del switch2 al switch1.

Figura 5.3: Prueba de conectividad entre los switches.

Antes de terminar esta sección, ya que tenemos las consolas abiertas desde SSH, vamos a simular que haremos como UniConfig, y entraremos con *vtysh* para leer configuración y editarla para alguna interfaz.

- Leeremos la configuración con el comando *show running-config*.
- Entraremos al modo de configuración con *configure terminal*. Después con el comando *interface Ethernet1* podremos editar algo de esa interfaz. Cambiamos la descripción con *description "my new description"*. Ahora cambiamos de interfaz con *exit* e *interface Ethernet2*, y apagamos esa con *shutdown*. Podemos ver la nueva configuración con *show running-config* otra vez.

Lo mostramos en la figura 5.4. El próximo paso será demostrar que podemos realizar acciones como estas con UniConfig.

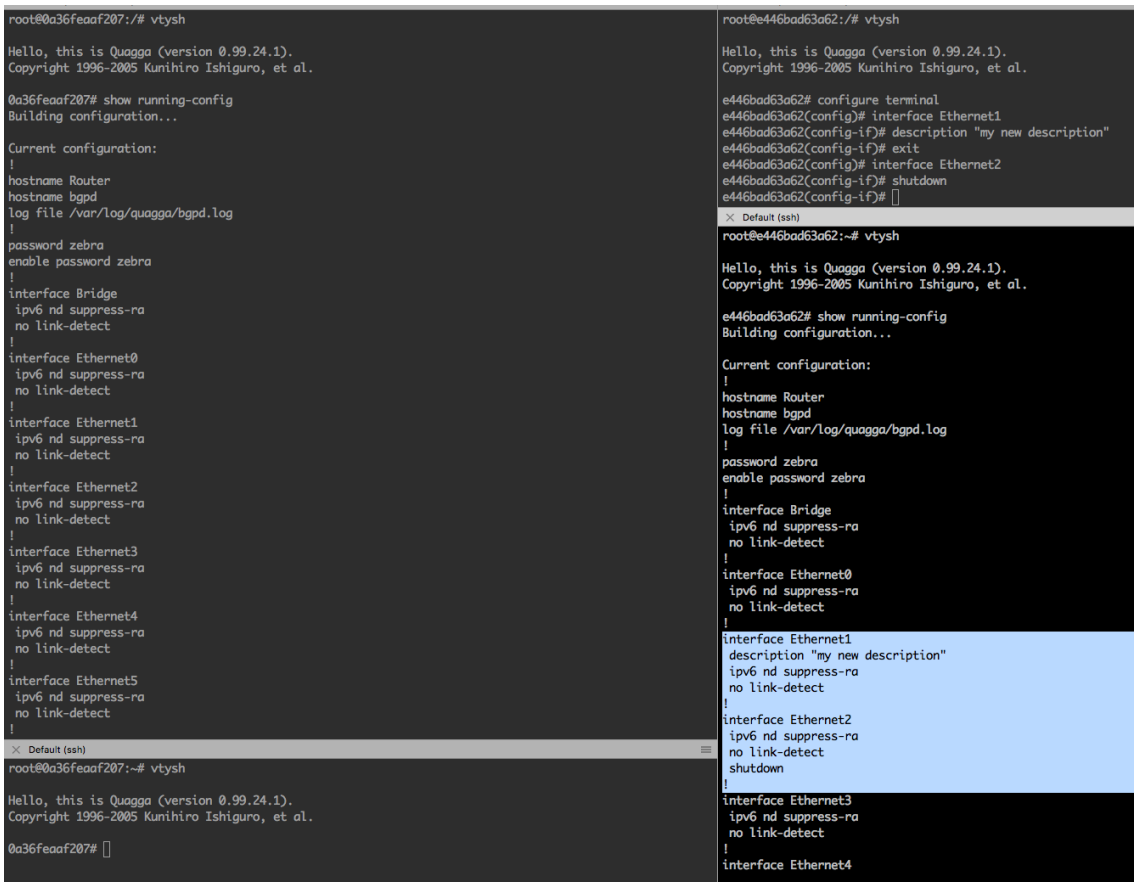


Figura 5.4: A la izquierda leemos la configuración actual. Arriba a la derecha modificamos Ethernet1 y Ethernet2. Abajo a la derecha volvemos a leer la configuración y vemos los cambios.

5.2. Demostración de la configurabilidad con FRINX en el entorno virtual

El primer paso para poder utilizar FRINX UniConfig es instalar las *cli-units*. Para eso, empleamos Maven como ya se ha mencionado en capítulos anteriores, por medio del comando:

```
mvn clean install -DskipTests -Dcheckstyle.skip \
-Dmaven.javadoc.skip=true -Djacoco.skip
```

Creamos el alias *frinxbuildfast* para este comando, y podemos mostrar el final de su ejecución en la figura 5.5.

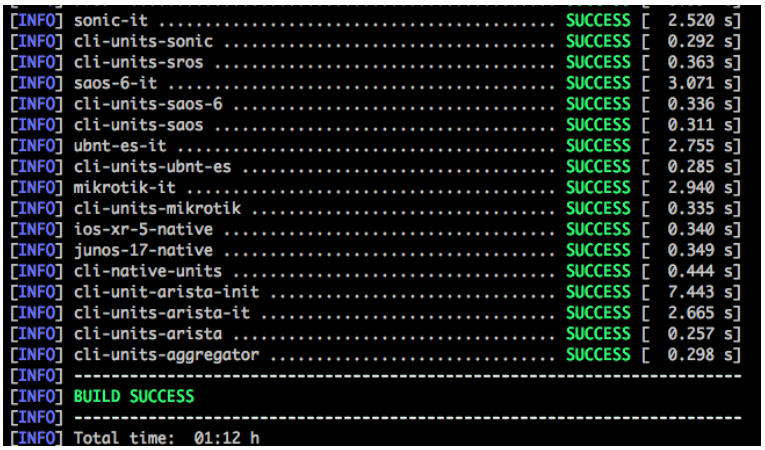


Figura 5.5: Parte final de la salida de la instalación del proyecto *cli-units* con Maven.

El siguiente paso es hacer lo propio con el otro repositorio, *distribution*, pues es el que tendrá el script que lanzará UniConfig. Por medio del comando:

```
cd ~/$PATH_TO_DISTRIBUTION/distribution/distribution-lighty-uniconfig; \
frinxbuildfast; cd target/; unzip uniconfig-4*; \
cd \$(ls | grep uniconfig | head -n 1);\
./run\_uniconfig.sh -l [frinx-license-secret-token] --debug
```

(para el cual crearemos otro alias: *frinxlighty*), instalaremos el proyecto y lanzaremos UniConfig (se necesita el token secreto de licencia de FRINX, obtenido como usuario de *FRINX.io*). Una vez termine, veremos en la salida los mensajes *Uniconfig layer ACTIVATED* y *Uniconfig layer initialized*.

Ya tenemos los proyectos listos, así que vamos a crear el entorno de SONiC tal y como hicimos en el capítulo anterior, por medio del script **start_ssh_sonic.sh** (usamos *root* como contraseña). Abrimos tres terminales. Desde la primera hemos ejecutado el script anterior. Desde la segunda, vamos a conectarnos al switch1 para escuchar con tcpdump la interfaz eth1, tal y como hicimos antes. Desde la última, vamos a conectarnos al switch2 y comenzar a hacer ping al switch1. Al igual que en el capítulo

anterior, comprobamos que esta comunicación funciona correctamente y lo vemos en la figura 5.6.

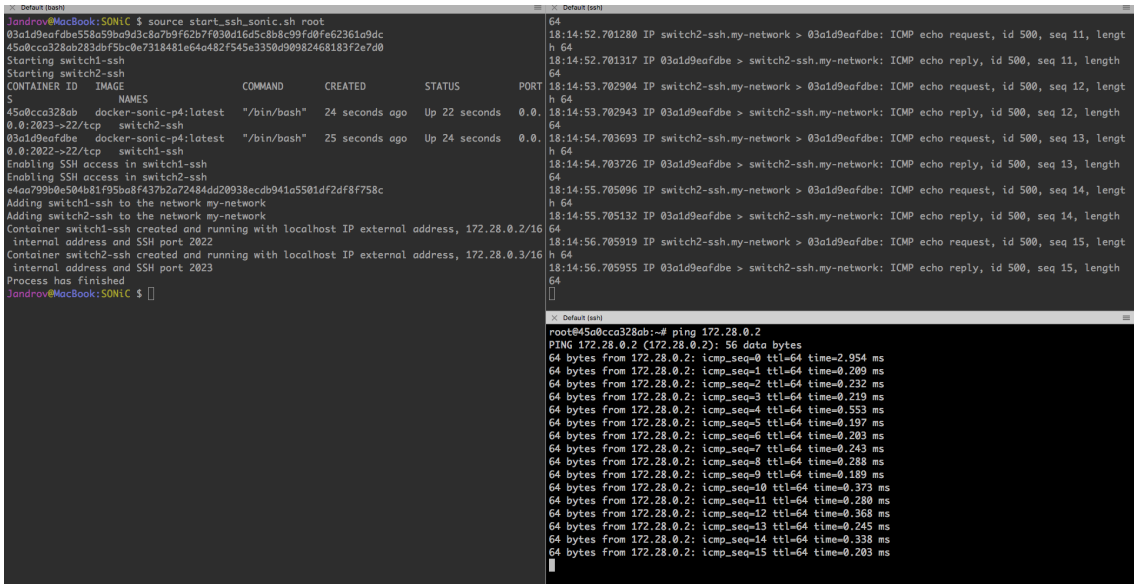


Figura 5.6: Switches lanzados y ya tenemos comunicación de switch2 a switch1.

Ahora vayamos a lo importante, tenemos que configurar el dispositivo con UniConfig. Para ello, enviaremos desde Postman unas peticiones HTTP al servidor donde tenemos corriendo UniConfig, el cual utilizará la cli-unit de SONiC implementada, para traducirla a comandos de CLI que permitan realizar nuestras operaciones. Estas peticiones las hemos obtenido de una colección de Postman de prueba que nos facilitó Šimon, las cuales hemos modificado para nuestro uso en SONiC. Se encuentran en nuestro repositorio de GitHub.

`https://github.com/Jandrov/SONiC/tree/master/material_extra/frinx_uniconfig`

Primero debemos enviar la petición que llamamos *mount switch1*, que inicializa el dispositivo. Usará la *init unit* implementada, y necesitaremos incluir en el cuerpo de dicha petición HTTP, el NOS que vamos a configurar (sonic), la IP y puerto SSH donde está corriendo el dispositivo (en este caso, localhost y 2022, pues es el switch1), y el usuario y contraseña (que hemos dicho que son root ambos). Veremos que la respuesta es correcta, y enviando después la petición que llamamos *check uniconfig status*, comprobaremos que el dispositivo está inicializado correctamente. En cuanto eso ocurra, podremos recuperar la configuración actual del dispositivo, tal y como podríamos ver con el comando *show running-config*.

El siguiente paso será enviar una petición para cambiar la descripción de la interfaz *eth1*. Para ello tendremos que enviar la petición que llamamos *config desc interface eth1*, indicando la interfaz que queremos modificar y la nueva descripción en el cuerpo. Una vez esté enviada, enviaremos la petición *commit* para aplicar esta nueva configuración. Podemos enviar entonces la petición que llamamos *get interface eth1 config*, para ver la configuración actual de eth1, que posee la nueva descripción añadida.

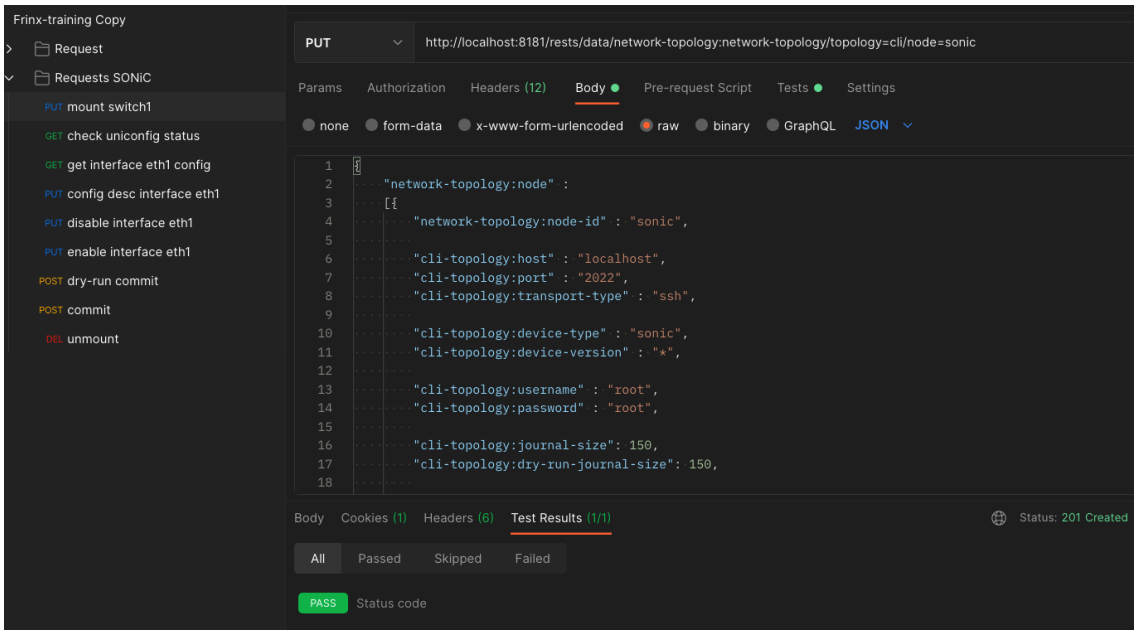


Figura 5.7: Cuerpo de la petición para inicializar SONiC.

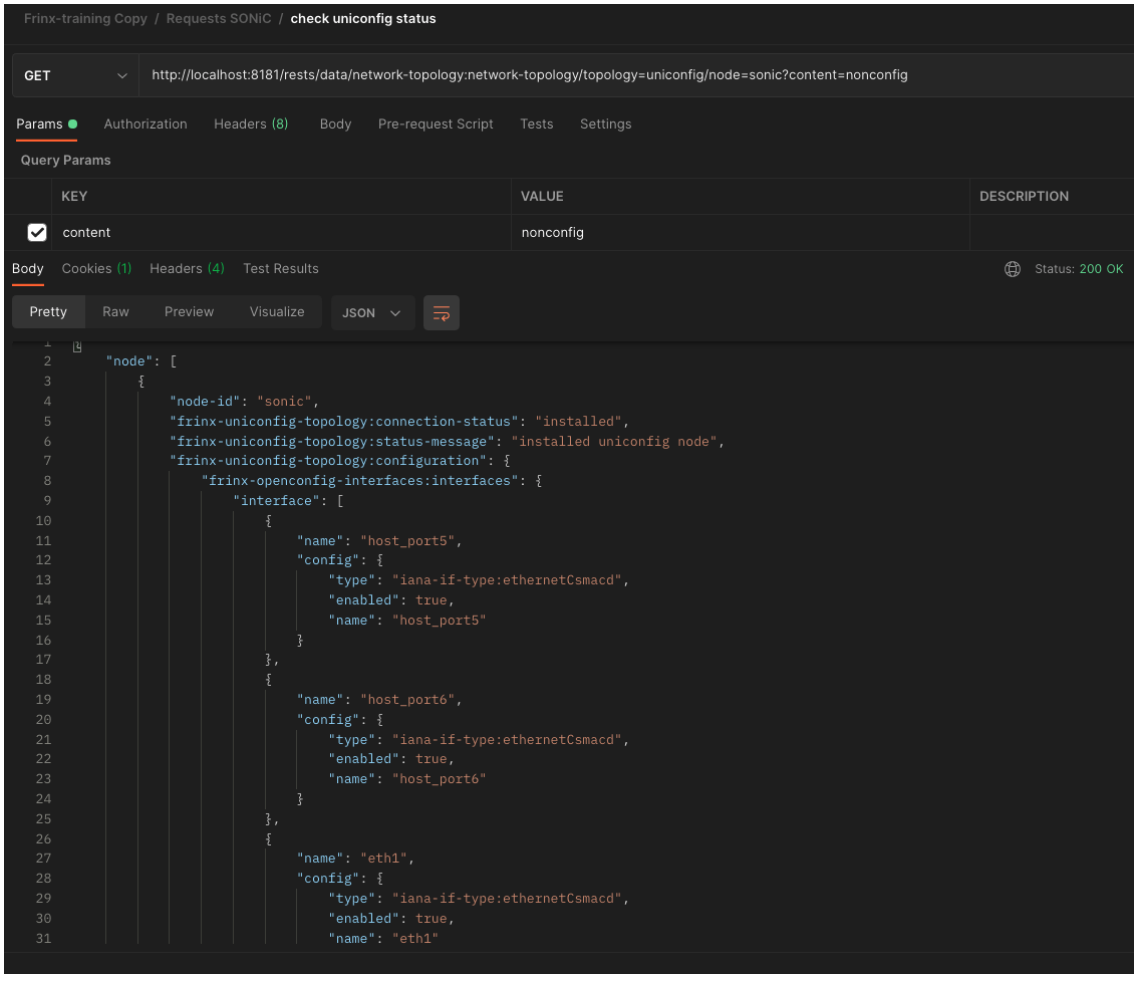


Figura 5.8: Respuesta con el estado de la configuración actual del dispositivo.

Figura 5.9: Cuerpo de la petición para deshabilitar la interfaz *eth1*.

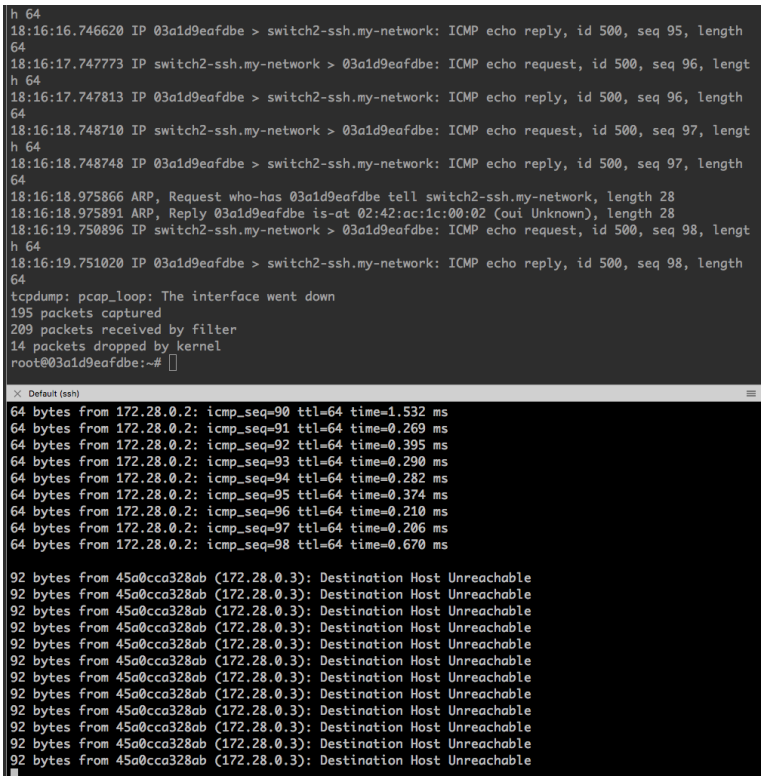


Figura 5.10: Comunicación rota entre los switches, debido a haberse apagado la interfaz *eth1*. Arriba vemos la ejecución cortada de tcpdump. Debajo vemos el mensaje *Destination Host Unreachable* de los pings.

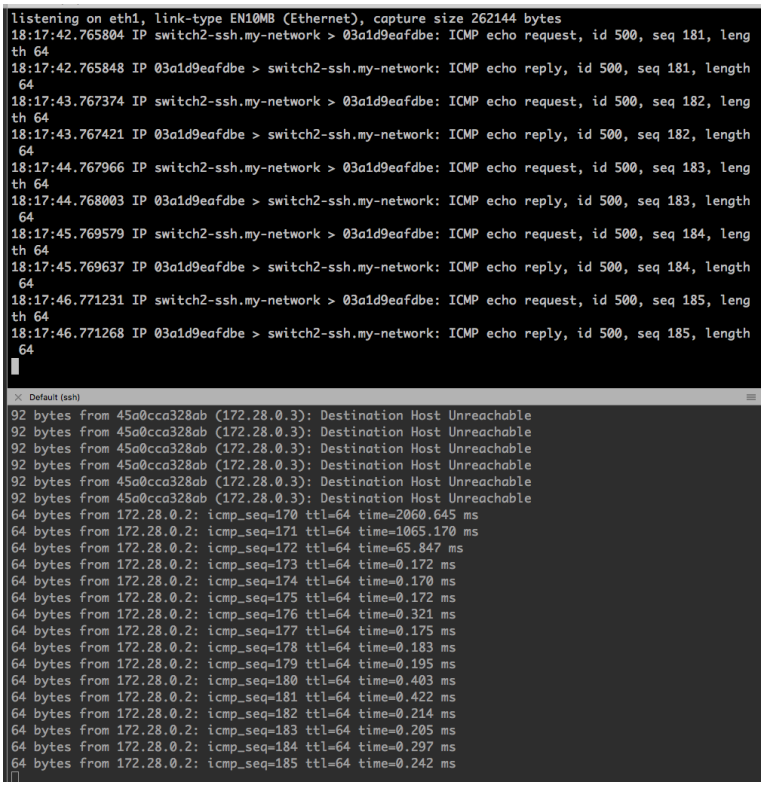


Figura 5.11: Comunicación restaurada entre los switches, al volver a encenderse la interfaz *eth1*.

Pero podemos comprobar que esto no altera la comunicación de nuestros switches, así que lo siguiente será enviar la petición que llamamos *disable interface eth1*, que es similar a la que cambiaba la descripción, pero cambiando el cuerpo. Ahora indicaremos *enabled: false*, así que cuando hagamos *commit*, veremos que los pings ya no llegan al switch1: la comunicación se ha cortado. De hecho, podemos ver que tcpdump termina de ejecutarse, ya que nos dice que la interfaz se ha deshabilitado. Esperamos unos segundos y comenzamos a ver mensajes en la terminal del ping, con el texto *Destination Host Unreachable*. Hemos logrado lo que buscábamos.

Para terminar, podemos ver otra vez con *get interface eth1 config* que aparece como deshabilitada la interfaz eth1, así que vamos a enviar la petición que llamamos *enable interface eth1*, para realizar la acción opuesta, activarla. Tras hacer el *commit*, podemos ver que los pings vuelven a llegar automáticamente al switch1, y podemos volver a ver con tcpdump que la comunicación existe de nuevo.

De este modo, hemos comprobado que nuestra unidad de traducción de CLI para SONiC **funciona correctamente**. En las figuras 5.7-5.11 ilustramos algunos de los pasos anteriores.

5.3. Contribución realizada al proyecto *OpenSource*

El resultado de la contribución realizada en el proyecto *OpenSource cli-units* de FRINX fue más que satisfactorio, ya que no solo se ha demostrado que la unidad funciona correctamente, sino que

además el código fue publicado el 6 de abril de 2021. Este es el commit.

Al terminar la contribución, se realizó una demostración sencilla para Gerhard Wieser, CEO de FRINX, el cual resultó satisfecho con el trabajo realizado. Dicha demostración se encuentra publicada en LinkedIn por FRINX.

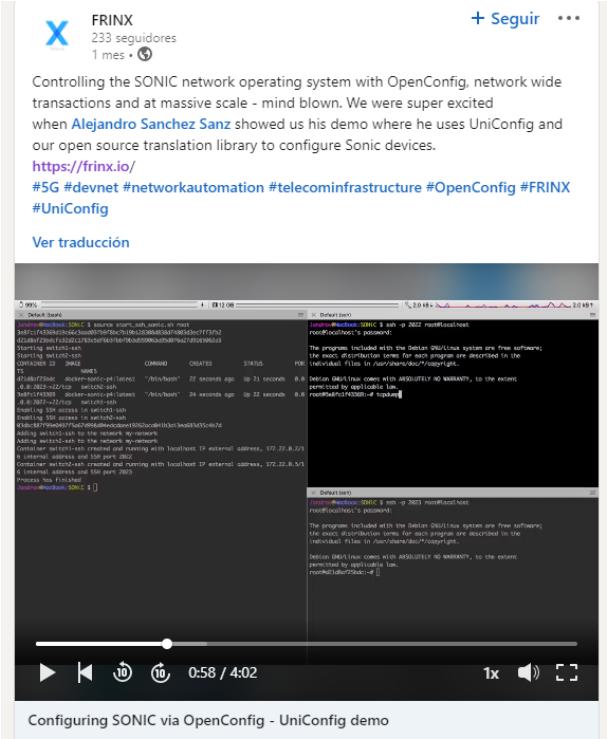


Figura 5.12: Publicación de FRINX en LinkedIn con su impresión sobre la demo realizada.

Por último, se plantearon posibles contribuciones posteriores para ampliar el trabajo realizado, que se comentarán en 6.2.

6

Conclusiones y trabajo futuro

6.1. Conclusiones

En este capítulo, vamos a revisar los objetivos definidos en la introducción para justificar que se han cumplido, explicando además los conocimientos y herramientas que han ayudado a su cumplimiento.

Los objetivos del trabajo definidos en la sección 1.2 eran los siguientes:

- Describir las soluciones basadas en tecnologías abiertas.
- Conocer herramientas para facilitar el desarrollo de interfaces abiertas.
- Desarrollar el entorno de pruebas para validar el funcionamiento.
- Demostrar el funcionamiento de la interfaz desarrollada.

Comenzamos este trabajo estudiando las tecnologías abiertas, que permitían la desagregación de los elementos de red. En particular, describimos las soluciones basadas en SDN y *White-Boxes*. Para el desarrollo de muchas de estas tecnologías, vimos que los proyectos *OpenSource* toman cada vez mayor importancia, existiendo muchos de ellos que permiten el desarrollo de interfaces abiertas. Para poder realizar una contribución en uno de estos proyectos, se crearon primero entornos virtuales donde trabajar con dos NOS: Goldstone y SONiC. Tras observar que el segundo era más prometedor, realizamos una contribución al proyecto *OpenSource* de FRINX UniConfig y ampliamos nuestro entorno de pruebas para validar su funcionamiento. Por último, probamos la unidad de traducción implementada y demostramos que funcionaba correctamente. Por tanto, vemos justificado el correcto cumplimiento de los objetivos planteados.

Merece la pena mencionar los conocimientos y herramientas aprendidos en algunas de las asignaturas de la carrera, pues han ayudado en gran medida a la realización de este trabajo. Podemos destacar Redes I y Redes II para comprender algunos conceptos del mundo de las telecomunicaciones, pero no son las únicas. Gracias a Sistemas Operativos se adquirieron conocimientos fundamentales para la creación de los entornos virtualizados basados en contenedores o en máquinas virtuales. Además, la contribución en UniConfig se hizo en lenguaje Java, el cual se aprendió en las asignaturas de Análisis y Diseño de Software, y Proyecto de Análisis y Diseño de Software. Finalmente, no podemos olvidar

la gran cantidad de prácticas realizadas en las distintas asignaturas de la carrera, cuyas largas horas de trabajo nos han permitido adquirir mucha experiencia a la hora de enfrentarnos a retos de cualquier tipo, y nos han ayudado a desarrollar una buena ética de trabajo, aplicada en este proyecto.

En conclusión, hemos podido explorar unas tecnologías punteras que en cierta medida desconocíamos y que no poseen demasiada documentación hasta el momento, por lo que la realización de este trabajo ha supuesto un gran reto a nivel personal. Sin embargo, el resultado ha sido satisfactorio, ya que se han cumplido los objetivos marcados y se han recibido comentarios positivos sobre la contribución realizada.

6.2. Trabajo futuro

Dado que estas tecnologías siguen en constante desarrollo y todavía son punteras, los posibles trabajos futuros que pueden ampliar nuestro proyecto, son variados. Algunos ejemplos podrían ser:

- Ampliación de la cli-unit de SONiC implementada para FRINX UniConfig: Como ya se mencionó, durante este proyecto hemos implementado algunas unidades, pero se podría ampliar la contribución por medio de la implementación de otras como la *acl unit* o la *bgp unit*. Incluso, se podría ampliar la *interface unit* realizada con submódulos para configurar las IPs o VLANs.
- Realización de pruebas en dispositivos físicos: En este trabajo hemos creado entornos virtuales y realizado ahí las pruebas, pero un posible paso futuro podría ser mover el entorno a uno con dispositivos físicos que tengan corriendo SONiC. Las pruebas de la cli-unit se podrían realizar ahí, y entonces necesitaríamos implementar la *it unit*, como se mencionó en la sección 4.2.

Bibliografía

[1] Microsoft Azure. SONiC P4 Software Switch. Última modificacion: 2017. Visitar.

[2] Samier Barguil, Víctor López, and Juan Pedro Fernández-Palacios Giménez. Towards an Open Networking Architecture. In *2020 International Conference on Optical Network Design and Modeling (ONDM)*, pages 1–3, 05 2020.

[3] A. Bierman, M. Bjorklund, and K. Watsen. RESTCONF Protocol. Enero 2017. Visitar.

[4] M. Bjorklund. The YANG 1.1 Data Modeling Language. Agosto 2016. Visitar.

[5] Cloudflare. ¿Qué es el modelo OSI? Último acceso: junio 2021. Visitar.

[6] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. Network Configuration Protocol (NETCONF). Junio 2011. Visitar.

[7] FRINX. FRINX: Fast network automation. Último acceso: junio 2021. Visitar.

[8] FRINX. Implementing CLI Translation Unit. Último acceso: junio 2021. Visitar.

[9] FRINX. UniConfig Components. Último acceso: junio 2021. Visitar.

[10] Javier Galán Sánchez. Verification and Validation Methodology for Interfaces in Network Environments. pages 1–39, 2019.

[11] Red Hat Inc. ¿Qué son las redes definidas por software? Último acceso: junio 2021. Visitar.

[12] Hyung-Soo Kim. Linux on Network Switch and Management. 2017. Visitar.

[13] Kubernetes. ¿Por qué usar contenedores? Última modificación: junio 16, 2020. Visitar.

[14] Víctor López. Software Defined Newtworking for Network Operators. pages 1–66, 11 2018.

[15] Víctor López, Wataru Ishida, Arturo Mayoral, Takafumi Tanaka, Óscar González de Dios, and Juan Pedro Fernández-Palacios. Enabling fully programmable transponder white boxes [Invited]. *IEEE/OSA Journal of Optical Communications and Networking*, 12(2):A214–A223, 2020.

[16] PalC Networks. OOPT NOS - Goldstone. 2020. Visitar.

[17] OpenConfig. OpenConfig: Vendor-neutral, model-driven network management designed by users. 2016. Visitar.

Acrónimos

ASIC *Application Specific Integrated Circuit*. Aplicaciones específicas para circuitos integrados.

CLI *Command-Line Interface*. Interfaz de línea de comandos.

CRUD *Create, Read, Update, Delete*. Se refiere a las funciones básicas sobre bases de datos.

HTTP *Hypertext Transfer Protocol*. Protocolo de Transferencia de Hipertexto.

HTTPS *Hypertext Transfer Protocol Secure*. Protocolo Seguro de Transferencia de Hipertexto.

IaaS *Infrastructure as a Service*. Infraestructura como servicio.

ICMP *Internet Control Message Protocol*. Protocolo de control de mensajes de internet.

IETF *Internet Engineering Task Force*. Grupo de trabajo de ingeniería de Internet.

JSON *JavaScript Object Notation*. Notación de objeto de JavaScript.

LAN *Local Area Network*. Red de área local.

NOS *Network Operating System*. Sistema operativo de red.

OCP *Open Computer Project*.

ONL *Open Network Linux*.

OOPT *Open Optical & Packet Transport*.

OSI *Open System Interconnection*. Modelo de interconexión de sistemas abiertos.

PaaS *Platform as a Service*. Plataforma como servicio.

RPC *Remote Procedure Call*. Llamada a procedimiento remoto.

SAI *Switch Abstraction Interface*. Interfaz de abstracción del conmutador de red.

SDN *Software Defined Networks*. Redes definidas por software.

SDNc *Software Defined Network controller*. Controlador de red definida por software.

SONiC *Software for Open Networking in the Cloud*. Software para redes abiertas en la nube.

SwSS *Switch State Service*. Servicio de estado del conmutador.

TAI *Transponder Abstraction Interface*. Interfaz de abstracción del transpondedor.

TCO *Total Cost of Ownership*. Coste total de propiedad.

TIP *Telecom Infra Project*.

VPN *Virtual Private Network*. Red privada virtual.

XML *Extensible Markup Language*. Lenguaje de marcado extensible.

YANG *Yet Another Next Generation*.


UAM
UNIVERSIDAD AUTONOMA
DE MADRID